# Inference Instructions and Notes

**Pre-trained models can be executed on unseen test data provided in the file `new_data.parquet`.** This file may be replaced in-place with any new test dataset, and predictions will automatically be generated using the `main.py` script.

For demonstration purposes, the accompanying notebook generates a 10-day test sample randomly selected from the existing `training_data.parquet` dataset. This test sample is saved as `new_data.parquet` with its dates modified to appear consecutively after the maximum date in the training data.

All model development and analysis are documented in the Jupyter notebook `230P-HW1-sol.ipynb`. Final models have been pre-trained using the methodology outlined in the notebook and serialized as `.pkl` files.

The inference pipeline expects input in the same structure as `training_data.parquet`, excluding the target column `ret`. If the `ret` column is present in `new_data.parquet`, it will be automatically dropped during inference to avoid data leakage.

# Feature Engineering Strategy

The dataset consists of five numerical features (`macro2`, `price`, `firm1`, `firm2`, `firm3`), one temporal feature (`date`), one categorical feature (`macro1`), and the target variable to be predicted (`ret`).

The primary objective of the feature engineering process was to construct a rich and informative set of predictors by systematically expanding the input space. This involved generating all meaningful pairwise interactions among numerical variables, as well as constructing various price-based ratios to capture relative relationships between firms and market-level variables.

Additional transformations, such as lagged features, rolling statistics (mean, std, min, max), and polynomial expansions (squared, cubed), were layered on top of these interactions to further enrich the feature space. The goal was to extract as much predictive signal as possible, even if it meant introducing a high-dimensional feature matrix.

The use of regularized models such as LASSO (L1) was motivated by their ability to perform automatic feature selection, effectively eliminating redundant or uninformative variables. This allowed for an aggressive feature engineering approach without the risk of overfitting due to high dimensionality.

# Pipeline Overview

The complete feature engineering pipeline is composed of five sequential substeps, each of which is modularized and applied through method chaining using the `.pipe()` interface. These steps are described below and elaborated in the subsequent subsections.

```python
def full_feature_engineering_pipeline(df):
    return (df.pipe(feature_interactions)
              .pipe(feature_one_hot_encoding)
              .pipe(feature_transformations)
              .pipe(rolling_features)
              .pipe(firm_id_one_hot_encoding))
```

Listing 1: Feature Engineering Pipeline

1. Feature Interaction Engineering
2. Categorical Feature One-Hot Encoding (`macro1`)
3. Feature Transformations (lagged, polynomial, log returns)
4. Rolling Window Statistics
5. Firm Identifier One-Hot Encoding (`firm_id`)

## 1. Feature Interaction Engineering

This step generates pairwise interaction terms across the core numerical features to uncover nonlinear relationships and capture cross-effects between firm-specific, macroeconomic, and price-related variables. In addition to multiplicative interactions (e.g., `firm1 * macro2`), the pipeline also includes price ratio features (e.g., `price / macro2`) which may capture relative scaling effects or valuation-like patterns.

```python
def feature_interactions(df):
    '''
    Creates interaction terms between firm variables, macro variables, and price.
    '''
    print('Creating feature interactions')

    # Firm-to-firm interactions
    df['firm1*firm2'] = df['firm1'] * df['firm2']
    df['firm2*firm3'] = df['firm2'] * df['firm3']
    df['firm1*firm3'] = df['firm1'] * df['firm3']

    # Firm-to-macro interactions
    df['firm1*macro2'] = df['firm1'] * df['macro2']
    df['firm2*macro2'] = df['firm2'] * df['macro2']
    df['firm3*macro2'] = df['firm3'] * df['macro2']

    # Firm-to-price interactions
    df['firm1*price'] = df['firm1'] * df['price']
    df['firm2*price'] = df['firm2'] * df['price']
    df['firm3*price'] = df['firm3'] * df['price']

    # Macro-to-price interaction
    df['macro2*price'] = df['macro2'] * df['price']

    # Price ratio features
    df['price/macro2'] = df['price'] / df['macro2']
    df['price/firm1'] = df['price'] / df['firm1']
    df['price/firm2'] = df['price'] / df['firm2']
    df['price/firm3'] = df['price'] / df['firm3']
```

```python
    return df
```

Listing 2: Feature Interaction Engineering

## 2. Categorical Feature One Hot Encoding (feature `macro1`)

To accommodate the categorical nature of the feature `macro1`, one-hot encoding was applied. This transformation enables the feature to be ingested by models such as Elastic Net Regression and XGBoost. While recent versions of XGBoost support native categorical handling, explicit one-hot encoding was performed in accordance with project instructions to ensure model-agnostic compatibility and transparency of the feature space.

```python

def feature_one_hot_encoding(df):
    '''
    One-hot encodes the 'macro1' column.
    '''
    print('One-hot encoding categorical feature: macro1')

    # Create dummy columns
    dummies = pd.get_dummies(df['macro1'], prefix='macro1', drop_first=False)

    # Drop original and concatenate dummies
    df = pd.concat([df.drop(columns=['macro1']), dummies], axis=1)
    return df
```

Listing 3: One Hot Encoding macro1

## 3. Feature Transformations

The next step involved applying non-linear transformations to the engineered features in order to capture potential higher-order relationships. These included:

- **Lag features:** Created per firm to capture short-term temporal dependencies.
- **Log returns:** Computed for the `price` feature to express relative changes.
- **Polynomial expansions:** Squared and cubed versions of key features were added to introduce curvature and interaction flexibility into the feature space.

Firm-wise computation was critical to preserve logical temporal structure and avoid data leakage across entities.

```
def feature_transformations(df):
    '''
    Adds lag, squared, cubed, and log-return transformations per firm.
    '''
    # Exclude macro1 from transformations
    macro1_features = [col for col in df.columns if col.startswith('macro1')]

    # Select columns to transform
    columns = [col for col in df.columns if col not in ['firm_id', 'date', 'ret'] +
    macro1_features]

    for firm_id in tqdm(df['firm_id'].unique(), desc='Feature transformations per
    firm_id'):
        firm_mask = df['firm_id'] == firm_id

        for col in columns:
            # Lag-1 feature
            df.loc[firm_mask, f'{col}_lag_1'] = df.loc[firm_mask, col].shift(1)

            # Polynomial features
            df[f'{col}_squared'] = df[col] ** 2
            df[f'{col}_cubed'] = df[col] ** 3

        # Log return for price (requires lag column to be generated first)
        df.loc[firm_mask, f'price_log_ret'] = (
            np.log(df.loc[firm_mask, 'price'] / df.loc[firm_mask, 'price_lag_1'])
        )

    return df
```

Listing 4: Feature Transformations

## 4. Rolling Features

To incorporate temporal context, rolling window statistics were computed over a 20-period horizon for all numeric features. The rolling features included:

- **Mean, Min, Max, Std:** Basic summary statistics within the window.
- **High/Low Ratio:** Max divided by Min — useful as a proxy for volatility skew.
- **Range:** Difference between Max and Min — an amplitude signal.
- **Window-based Return:** Percentage return over the lag window.

These features help embed autocorrelation and volatility structure that may aid the model in predicting the target variable more effectively.

```
def rolling_features(df, window_list=[20]):
    '''
    Adds rolling mean, std, min, max, range, and returns for each numeric feature
    per firm.
    '''
```

```python
    # Identify macro features to exclude from rolling calculations
    macro1_features = [col for col in df.columns if col.startswith('macro1')]

    # Select all numeric columns excluding identifiers and macro1
    columns = [col for col in df.columns if col not in ['firm_id', 'date', 'ret'] + macro1_features]

    # Iterate over each unique firm
    for firm_id in tqdm(df['firm_id'].unique(), desc='Rolling features per firm_id'):
        for col in columns:
            firm_mask = df['firm_id'] == firm_id
            for window in window_list:
                # Rolling average
                df.loc[firm_mask, f'{col}_rolling_avg_{window}'] = df.loc[firm_mask, col].rolling(window).mean()

                # Rolling standard deviation
                df.loc[firm_mask, f'{col}_rolling_std_{window}'] = df.loc[firm_mask, col].rolling(window).std()

                # Rolling max
                df.loc[firm_mask, f'{col}_rolling_max_{window}'] = df.loc[firm_mask, col].rolling(window).max()

                # Rolling min
                df.loc[firm_mask, f'{col}_rolling_min_{window}'] = df.loc[firm_mask, col].rolling(window).min()

                # High/low ratio (max / min)
                df.loc[firm_mask, f'{col}_rolling_high/low_{window}'] = (
                    df.loc[firm_mask, f'{col}_rolling_max_{window}'] /
                    df.loc[firm_mask, f'{col}_rolling_min_{window}']
                )

                # Range = max - min
                df.loc[firm_mask, f'{col}_rolling_range_{window}'] = (
                    df.loc[firm_mask, f'{col}_rolling_max_{window}'] -
                    df.loc[firm_mask, f'{col}_rolling_min_{window}']
                )

                # Window-based return (current / lag - 1)
                df.loc[firm_mask, f'{col}_rolling_ret_{window}'] = (
                    df.loc[firm_mask, col] / df.loc[firm_mask, col].shift(window - 1) - 1
                )
    return df
```

Listing 5: Rolling Features

## 5. Firm ID One Hot Encoding

To complete the pipeline, the `firm_id` column—representing a categorical identifier of the security—was also one-hot encoded. Since the numeric values of identifiers do not imply ordinal relationships, encoding was necessary to prevent the model from misinterpreting them. This process mirrors the transformation applied to `macro1`, ensuring consistency across all categorical inputs.

```
def firm_id_one_hot_encoding(df):
    '''
    One-hot encodes the 'firm_id' column.
    '''
    print('One-hot encoding firm_id')

    # Create dummy columns
    dummies = pd.get_dummies(df['firm_id'], prefix='firm_id', drop_first=False)

    # Drop original and concatenate dummies
    df = pd.concat([df.drop(columns=['firm_id']), dummies], axis=1)
    return df
```

Listing 6: Firm ID One Hot Encoding

# ElasticNet

## 1. Baseline Elastic Net Model

- Fixed hyper-parameters (selected via the initial validation-set grid search):

$$\alpha = 0.01, \qquad \ell_1\text{-ratio} = 0.10.$$

- Model fitted on the standardized training matrix and evaluated on the chronological validation window.

**Impact of the engineered feature set**
Augmenting the original macro and firm fundamentals with the engineered interaction/momentum features measurably boosts generalization:

| **Feature Set** | **Validation MSE** | **Validation $R^2$** |
|---|---|---|
| Original features only | 0.0023 | 0.5852 |
| With engineered features | 0.0017 | 0.6963 |

Table 1: Effect of engineered features on baseline Elastic Net performance.

**Interpretation**

- **Error reduction.** The validation MSE drops by $\approx$ 26% (from $2.3 \times 10^{-3}$ to $1.7 \times 10^{-3}$), indicating that the additional interactions capture systematic variation previously left in the residuals.
- **Explained variance.** Validation $R^2$ climbs from 0.585 to 0.696, a relative improvement of roughly 19%, confirming that the model accounts for nearly 70

## 2. Hyper Parameter Tuning

1. **Initial grid search (validation-set driven)**

- Swept $\sim 40$ logarithmically spaced $\alpha$ values ($10^{-2}-10^{0}$) against 50 uniformly spaced $\ell_1$ ratios ($0.10-1.00$).
- For each $(\alpha, \ell_1)$ pair the model was fit on the *training split* and scored on the *held-out validation split* using MSE / $R^2$.
- The best-performing pair on the validation set was refit on the full training data.
- *Caveat:* Because every candidate is judged on a single validation fold, the winner is effectively tuned to that fold and may look overly optimistic on unseen data.

```python
    def hyperopt_elasticnet():
        our_train = pd.read_parquet('./train+features.parquet')
        train, val = temporal_train_val_split(our_train, cutoff_frac=0.8)
        train = train.dropna()

        X_train, y_train = prepare_X_y(train, drop_cols=['date'])
        X_val, y_val = prepare_X_y(val, drop_cols=['date'])
        X_train_scaled, X_val_scaled, _ = standardize_with_train_stats(X_train, X_val)

        all_alpha = np.logspace(-2, 0, 40)
        all_l1 = np.linspace(0.1, 1, 50)

        results = {}
        for alpha in all_alpha:
            print(f"Testing alpha: {alpha}")
            for l1 in tqdm(all_l1, total=len(all_l1)):
            # for l1 in all_l1:
                # print(f"Testing l1_ratio: {l1}")
                # Fit ElasticNet with current parameters
                elastic_net = ElasticNet(alpha=alpha, l1_ratio=l1, max_iter=10_000, random_state=42)
                elastic_net.fit(X_train_scaled, y_train)
                y_pred = elastic_net.predict(X_val_scaled)
                mse = mean_squared_error(y_val, y_pred)
                r2 = r2_score(y_val, y_pred)

                # print(f"ElasticNet alpha: {alpha}, l1_ratio: {l1}, MSE: {mse:.4f}, R2: {r2:.4f}")
                results[(alpha, l1)] = (mse, r2)

        results = sorted(results.items(), key=lambda x: x[1][1])

        # Best parameters
        best_alpha, best_l1 = results[-1][0]
        elastic_net = ElasticNet(alpha=best_alpha, l1_ratio=best_l1, max_iter=10_000, random_state=42)
        elastic_net.fit(X_train_scaled, y_train)
        y_pred = elastic_net.predict(X_val_scaled)
        mse = mean_squared_error(y_val, y_pred)
        r2 = r2_score(y_val, y_pred)

        print(f"ElasticNet best alpha: {best_alpha}")
```

```python
    print(f"ElasticNet best l1_ratio: {best_l1}")
    print(f"Validation MSE: {mse:.4f}")
    print(f"Validation R2: {r2:.4f}")

hyperopt_elasticnet()

```

Listing 7: Grid Search CV

2. **Cross-validated tuning inside the training set**
   - Replaced the manual grid search with `ElasticNetCV`, retaining the same candidate grid.
   - Performed $k$=5-fold cross-validation *entirely within the training split*.
   - Selected the $(\alpha, \ell_1)$ pair maximizing the mean CV score, reducing leakage of validation information.

```python
def elastic_net_cross_val():
    our_train = pd.read_parquet('./train+features.parquet')
    train, val = temporal_train_val_split(our_train, cutoff_frac=0.8)
    train = train.dropna()

    X_train, y_train = prepare_X_y(train, drop_cols=['date'])
    X_val, y_val = prepare_X_y(val, drop_cols=['date'])

    pipe = make_pipeline(
        StandardScaler(),                    # fitted on *each* train-fold
        ElasticNet(max_iter=10_000)
    )

    param_dist = {
        "elasticnet__alpha": np.logspace(-2, 0, 60),
        "elasticnet__l1_ratio": np.linspace(0.1, 1, 101),
    }


    inner_cv = TimeSeriesSplit(n_splits=5)

    search = RandomizedSearchCV(
        pipe,
        param_distributions = param_dist,
        n_iter       = 200,
        scoring      = "neg_mean_squared_error",
        cv           = inner_cv,
        n_jobs       = -1,
        random_state= 0,
        verbose      = 10,
    )

    #  --- guarantee an ASCII temp folder for Windows / joblib
----------------
    tmp_root = pathlib.Path(r"C:\tmp\joblib")
    tmp_root.mkdir(parents=True, exist_ok=True)
    os.environ["JOBLIB_TEMP_FOLDER"] = str(tmp_root)
    os.environ["OMP_NUM_THREADS"]    = "1"     # 1 BLAS thread per worker

    search.fit(X_train, y_train)
```

```python
        print("Selected params :", search.best_params_)
        y_pred = search.predict(X_val)
        mse_search = mean_squared_error(y_val, y_pred)
        r2_search = r2_score(y_val, y_pred)
        print(f"Search best alpha: {search.best_params_['elasticnet__alpha']}")
        print(f"Search best l1_ratio: {search.best_params_['
    elasticnet__l1_ratio']}")
        print(f"Search Validation MSE: {mse_search:.4f}")
        print(f"Search Validation R2: {r2_search:.4f}")

    elastic_net_cross_val()

```

Listing 8: Cross-Validation

3. **Model comparison**
   - Both tuned models were evaluated on:
     3.1. their respective *training data* (in-sample fit), and
     3.2. the external *validation split* (generalization).
   - The vanilla grid-search model exhibited a larger train–validation performance gap (higher train $R^2$, lower val $R^2$), signaling overfitting.
   - The CV-tuned model showed more consistent metrics across splits, indicating better generalization.

## Selected Hyper-parameters

| Hyper-parameter | Grid Search (val set) | RandomizedSearchCV (5-fold CV) |
|---|---|---|
| $\alpha$ | 0.0100 | 0.01169 |
| $\ell_1$ **-ratio** | 0.2286 | 0.5950 |

Table 2: Best Elastic Net configurations under two tuning schemes.

### Interpretation

- **Regularization strength ($\alpha$).** Both searches converge on a small—but non-negligible—$\alpha \approx 0.01$, implying that mild overall shrinkage is beneficial: it tempers coefficient variance without washing out signal.
- **Mixing parameter ($\ell_1$-ratio).**
  - *Grid search:* $\ell_1$=0.23 leans heavily toward ridge-like ($\ell_2$) regularization; coefficients are shrunk but only a few are driven exactly to zero.
  - *CV search:* $\ell_1$=0.60 shifts the balance toward lasso-like sparsity, zeroing more weak predictors while still retaining an $\ell_2$ component to stabilise correlated features.

## 4. Feature Importance

### Interpretation of Permutation Feature Importance (with Significance)

1. **Macro shock & persistence — `macro2` and `macro2_lag_1`**
   Shuffling `macro2` increases the validation–set MSE by roughly $2 \times 10^{-3}$, with its one–period lag close behind ($1.3 \times 10^{-3}$). Both $p$-values are $< 10^{-45}$, so we can be extremely confident that the model relies on the current and recent macro environment for accuracy.

2. **Firm–specific fundamentals — `firm2`, `firm1`, `firm3` and their lags**
   Each firm variable registers a smaller but still highly significant drop in performance when permuted ($p < 10^{-35}$). This confirms that cross–sectional differences among firms add signal beyond the overarching macro backdrop.

3. **Cross–effects and price dynamics — `price_log_ret`, `firm1*macro2`, `price/macro2`$^2$ `rolling_high/low_20`**
   The *interaction* `firm1*macro2` and the pure market feature `price_log_ret` both survive the 5% significance test, suggesting that macro conditions modulate firm impact and that short–horizon price momentum contributes incremental information. In contrast, higher–order engineered variants (e.g. `firm2*macro2_cubed_rolling_min_20`) yield zero mean importance and $p = 1$, indicating redundancy; these can be pruned to simplify the feature set without sacrificing predictive power.

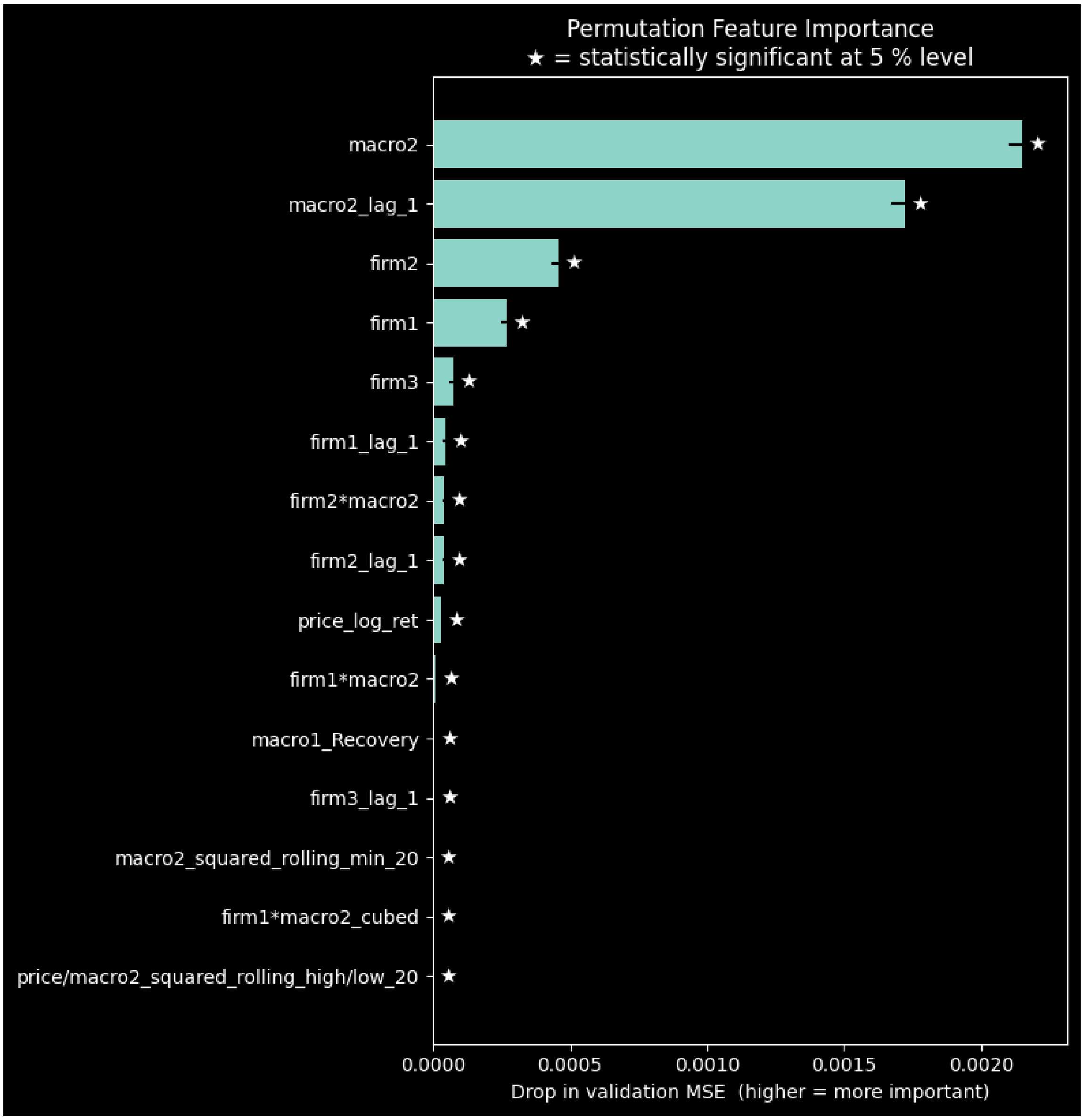
Overall, the model's predictive edge is anchored in macro shocks, refined by firm–level fundamentals, and sharpened by select interaction and price–momentum terms. Features with zero importance and non–significant $p$-values are prime candidates for removal, streamlining future iterations and potentially reducing over–fitting.

#### Interestingly, `FirmID` itself is not useful

The **FirmID indicator variables** (one–hot or label encodings that merely tag each entity) show **zero mean importance and** $p = 1.0$. This tells us that, once the model already sees:

- firm–level quantitative fundamentals (`firm1`, `firm2`, ...),
- macro context plus interaction terms,

**knowing the raw identity of the firm adds no extra predictive power**. In practical terms, the model distinguishes firms through their observable metrics rather than their labels; any firm–specific fixed effects are either weak or already captured by those continuous fundamentals.


Permutation Feature Importance
★ = statistically significant at 5 % level
macro2
macro2_lag_1
firm2
firm1
firm3
firm1_lag_1
firm2*macro2
firm2_lag_1
price_log_ret
firm1*macro2
macro1_Recovery
firm3_lag_1
macro2_squared_rolling_min_20
firm1*macro2_cubed
price/macro2_squared_rolling_high/low_20
0.0000
0.0005
0.0010
0.0015
0.0020
Drop in validation MSE (higher = more important)

# XGBoost

## 1. Baseline XGBoost Model

- `XGBRegressor` with moderate complexity:

$$\text{n\_estimators} = 200,\ \text{max\_depth} = 3,\ \eta = 0.10,\ \text{subsample} = 1,\ \text{colsample\_bytree} = 0.8.$$

- Fit on the scaled training matrix; evaluated on the validation window.

**Impact of the engineered feature set**
Adding the newly created interaction and momentum features noticeably improves generalization, shrinking validation MSE and lifting $R^2$:

| **Feature Set** | **Validation MSE** | **Validation $R^2$** |
|---|---|---|
| Original features only | 0.0025 | 0.5598 |
| With engineered features | 0.0018 | 0.6887 |

Table 3: Effect of engineered features on baseline XGBoost performance.

## 2. Hyper-parameter Tuning

- **Search space.** max_depth $\in \{4, 6\}$, $\eta \in \{0.05, 0.10\}$, $\gamma \in \{0, 0.1\}$, $\lambda \in \{0, 1, 5\}$.
- **Methodology.** *RandomizedSearchCV* with $k = 5$-fold cross-validation nested completely *inside* the training split. Mean squared error (negative) is the scoring metric.

## 3. Selected Configuration

$$\begin{array}{rr} \text{n\_estimators} = 1000 & \eta = 0.05 \\ \text{max\_depth} = 4 & \text{subsample} = 0.8 \\ \text{colsample\_bytree} = 0.8 & \gamma = 0 \\ \text{reg\_lambda} = 1.0 & \text{min\_child\_weight} = 1 \end{array}$$

**Rationale**

- $\eta = 0.05$ with n_estimators = 1000: a lower learning rate paired with more boosting rounds allows the model to make finer, more stable updates.
- max_depth = 4: deep enough to capture non-linear macro–firm interactions while still limiting model variance.
- subsample = 0.8 and colsample_bytree = 0.8: row- and column-subsampling inject bootstrap-style diversity, further curbing over-fitting.
- $\gamma = 0$: no additional split-gain threshold was needed once the other regularization levers were tuned.
- $\lambda = 1.0$ ($\ell_2$ regularization) and min_child_weight = 1 place mild penalties on overly large leaf weights, balancing bias and variance without suppressing meaningful signals.

## 4. Importance Score

| Rank | Feature | Split-count | Sig. |
|---|---|---|---|
| 1 | **firm2** | $\approx$ **63** | $\star$ |
| 2 | **macro2** | $\approx$ 60 | $\star$ |
| 3 | **macro2_lag_1** | $\approx$ 55 | $\star$ |
| 4 | **firm1** | $\approx$ 49 | $\star$ |
| 5 | **firm3** | $\approx$ 34 | $\star$ |
| 6 | **firm2_lag_1** | $\approx$ 28 | $\star$ |
| 7 | **firm2** $\times$ **macro2** | $\approx$ 25 | $\star$ |
| 8–10 | **macro1_Contraction, firm1_lag_1, price_log_ret** | 19–22 | $\star$ |
| 11–20 | Remaining terms | 10–19 | $\star$ |

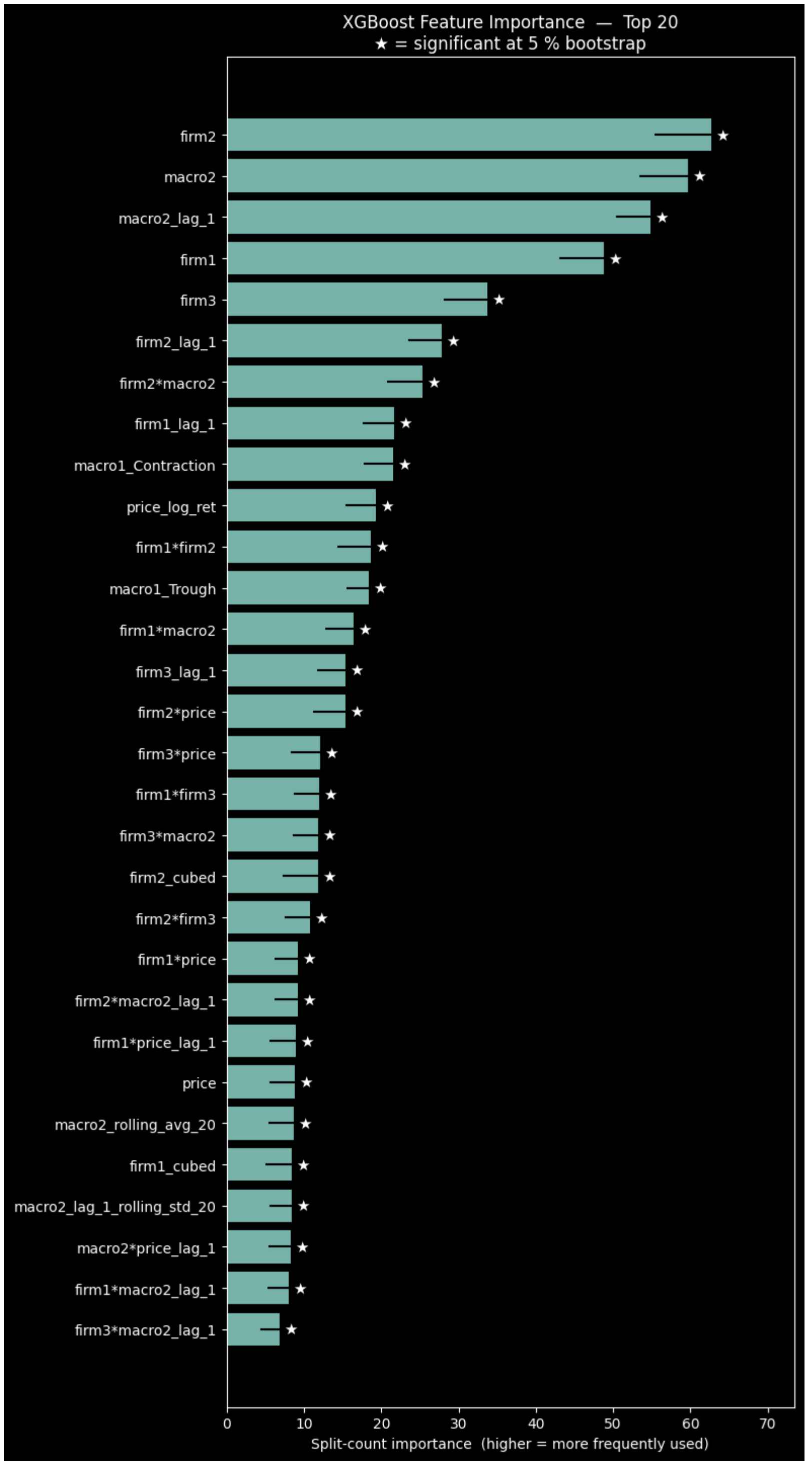
**Note:** Every feature in the Top 20 passes the bootstrap significance test ($\star$).

Firm-ID dummies never make the top list; their mean split-count is zero and $p = 1$, matching earlier conclusions.

### Key Takeaways

- **Macro shocks (`macro2`, lag) remain indispensable** across both models, underscoring their foundational predictive power.
- **Nonlinear modelling boosts firm fundamentals.** XGBoost uncovers interaction-heavy relationships that Elastic-Net (restricted to additive linear terms) cannot exploit, elevating `firm2`, `firm1`, and their lagged or interacted variants.
- **Complex engineered features are now useful.** Tree ensembles harness cubic or rolling interactions that previously appeared redundant, hinting at higher-order effects worth retaining in nonlinear pipelines.
- **FirmID columns stay redundant** and can be safely pruned.

Overall, while the linear model captures the broad macro signal, XGBoost extracts richer, context-dependent patterns — especially those linking firm-level variables to macro states and recent price dynamics — delivering a more nuanced importance landscape.


XGBoost Feature Importance — Top 20
★ = significant at 5 % bootstrap
firm2
macro2
macro2_lag_1
firm1
firm3
firm2_lag_1
firm2*macro2
firm1_lag_1
macro1_Contraction
price_log_ret
firm1*firm2
macro1_Trough
firm1*macro2
firm3_lag_1
firm2*price
firm3*price
firm1*firm3
firm3*macro2
firm2_cubed
firm2*firm3
firm1*price
firm2*macro2_lag_1
firm1*price_lag_1
price
macro2_rolling_avg_20
firm1_cubed
macro2_lag_1_rolling_std_20
macro2*price_lag_1
firm1*macro2_lag_1
firm3*macro2_lag_1
0
10
20
30
40
50
60
70
Split-count importance (higher = more frequently used)

| Aspect | Elastic-Net | XGBoost |
|---|---|---|
| **Dominant driver** | `macro2` (and its lag) lead the ranking | `firm2` edges out macro variables, with `macro2` & its lag still highly important |
| **Firm fundamentals** | Present but behind macro2; linear contribution only | Stronger presence (firm2, firm1, firm3 + lags) — boosted by nonlinear interactions |
| **Interactions & higher-order terms** | Few survive significance (`firm1` $\times$ `macro2`, `price_log_ret`) | Many more interaction terms gain importance, indicating XGBoost's capacity to model complex feature interplay |
| **Macro-cycle dummies** | Insignificant | `macro1_Trough`, `macro1_Contraction` emerge as significant, suggesting state-dependent effects the linear model misses |
| **FirmID indicators** | No importance | Still none — confirms they add no incremental value once quantitative fundamentals are present |

Table 5: Comparison of feature importance between Elastic-Net and XGBoost models