In [4]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.impute import SimpleImputer
from sklearn.dummy import DummyRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Lasso
from sklearn.linear_model import LassoCV
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import ElasticNet
from sklearn.linear_model import ElasticNetCV
from sklearn.kernel_approximation import RBFSampler
from sklearn.pipeline import make_pipeline
from sklearn.cross_decomposition import PLSRegression
from sklearn.ensemble import GradientBoostingRegressor
```

Question 2

(a) Rank-sorting Characteristics and Sharpe Ratios

Rank-sort each characteristic monthly to form characteristic-based portfolios.

Compute and plot the annualized Sharpe ratios of these portfolios.

In [8]:
```
# Load the data
df = pd.read_parquet('largeml.pq')
```

In [10]:
```
# Inspect the structure
print(df.shape)
print(df.columns)
print(df.index.names)
```

```
(79146, 212)
Index(['permno', 'yyyymm', 'AM', 'AOP', 'AbnormalAccruals', 'Accruals',
       'AccrualsBM', 'Activism1', 'Activism2', 'AdExp',
       ...
       'roaq', 'sfe', 'sinAlgo', 'skew1', 'std_turn', 'tang', 'zerotrade12
M',
       'zerotrade1M', 'zerotrade6M', 'ret'],
      dtype='object', length=212)
[None]
```

In [12]:
```
# Set multi-index using 'yyyymm' (monthly date) and 'permno' (firm ID)
df = df.set_index(['yyyymm', 'permno'])
print(df.index.names)
```

```
['yyyymm', 'permno']
```

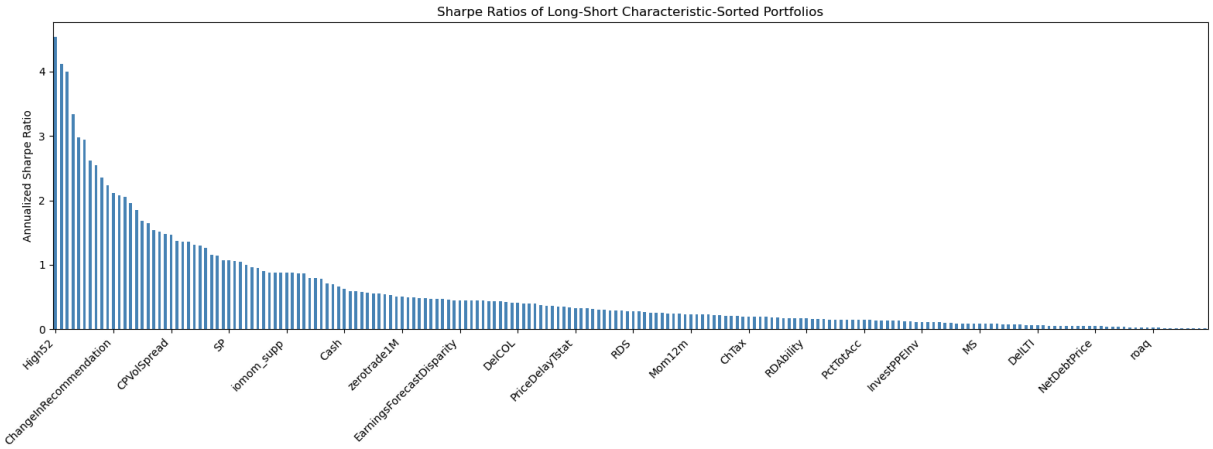
In [15]:
```
print(df['ret'].dtype)
df['ret'] = pd.to_numeric(df['ret'], errors='coerce')
```

```
object
```

In our long-short portfolio construction, we sort stocks by a given characteristic each month and go long the top 10% and short the bottom 10%. However, since we don't know the economic interpretation of characteristics, sometimes for some characteristics, a higher value may actually predict lower future returns. Thus, the long-short portfolio formed in the default direction may produce a negative Sharpe ratio. This does not necessarily mean the characteristic is useless — rather, it may indicate that the predictive signal is in the opposite direction. So we interpret negative Sharpe ratios as an opportunity to reverse the portfolio weights — going short the top and long the bottom. Therefore, we focus on the magnitude of the Sharpe ratio (using the absolute value) as a measure of the characteristic's predictive power, regardless of direction.

In [37]:
```
# Step 1: Extract characteristics
characteristics = [col for col in df.columns if col != 'ret']
monthly_returns_ls = {}

# Step 2: Loop through characteristics to compute long-short returns
for char in characteristics:
    def long_short(month):
        data = month[[char, 'ret']].dropna()
        if len(data) < 10:
            return np.nan

        # Define deciles
        top_thresh = data[char].quantile(0.9)
        bottom_thresh = data[char].quantile(0.1)

        # Long and short returns
        long_ret = data[data[char] >= top_thresh]['ret'].mean()
        short_ret = data[data[char] <= bottom_thresh]['ret'].mean()

        return long_ret - short_ret  # long-short return

    port_ret = df.groupby(level='yyyymm').apply(long_short)
    monthly_returns_ls[char] = port_ret

# Step 3: Combine into DataFrame
portfolio_returns = pd.DataFrame(monthly_returns_ls)

# Step 4: Compute annualized Sharpe ratios
def annualized_sharpe(series):
    mean = series.mean()
    std = series.std()
    if std < 1e-6:  # Avoid divide-by-zero
        return np.nan
    return (mean / std) * np.sqrt(12)

sharpe_ratios = portfolio_returns.apply(annualized_sharpe).dropna().abs().sc
# Step 5: Plot
```

```
plt.figure(figsize=(16, 6))
ax = sharpe_ratios.plot(kind='bar', color='steelblue')

plt.ylabel('Annualized Sharpe Ratio')
plt.title('Sharpe Ratios of Long-Short Characteristic-Sorted Portfolios')

# Show only every 10th label for readability
N = 10
xticks = ax.get_xticks()
xticklabels = sharpe_ratios.index

ax.set_xticks(xticks[::N])
ax.set_xticklabels(xticklabels[::N], rotation=45, ha='right')

plt.tight_layout()
plt.show()

# Step 6: Print best and worst performers
print("Top 5 Characteristics by Sharpe Ratio:")
print(sharpe_ratios.head())

print("\nBottom 5 Characteristics by Sharpe Ratio:")
print(sharpe_ratios.tail())
```




```
Top 5 Characteristics by Sharpe Ratio:
High52                 4.535245
ReturnSkew3F           4.116495
ReturnSkew             3.996626
AnnouncementReturn     3.340106
retConglomerate        2.984710
dtype: float64

Bottom 5 Characteristics by Sharpe Ratio:
VolumeTrend         0.019332
MomOffSeason        0.018558
BetaLiquidityPS     0.017946
MRreversal          0.016682
ChAssetTurnover     0.012666
dtype: float64
```

2(b) ML Methods

In [38]:
```
# Prepare X (characteristics) and y (shifted returns)
if 'permno' in df.columns:
    df = df.drop(columns=['permno'])
X = df.drop(columns=['ret'])  # All characteristics
y = df['ret']
```

In [39]:
```
# Create a 'date' variable for splitting, pull out the date part of the inde
df = df.copy()
df['yyyymm'] = df.index.get_level_values('yyyymm')

# Convert to datetime format
df['date'] = pd.to_datetime(df['yyyymm'].astype(str), format='%Y%m')

# Add back to X/y for splitting
X['date'] = df['date']
y = y.copy()
```

In [40]:
```
# Split train / val / test
n_train = 12 * 20
n_val = 12 * 12

months = sorted(df.index.get_level_values('yyyymm').unique())
months = pd.to_datetime(np.array(months).astype(str), format='%Y%m')

train_end = months[n_train - 1]
val_end = months[n_train + n_val - 1]

X_train = X[X['date'] <= train_end].drop(columns='date')
X_val   = X[(X['date'] > train_end) & (X['date'] <= val_end)].drop(columns='
X_test  = X[X['date'] > val_end].drop(columns='date')

y_train = y[X['date'] <= train_end]
y_val   = y[(X['date'] > train_end) & (X['date'] <= val_end)]
y_test  = y[X['date'] > val_end]
```

In [41]:
```
# Backup original data before cleaning for OLS, Lasso, Ridge, etc.
X_train_raw = X_train.copy()
X_val_raw = X_val.copy()
X_test_raw = X_test.copy()
y_train_raw = y_train.copy()
y_val_raw = y_val.copy()
y_test_raw = y_test.copy()
```

In [42]:
```
# Initialize the dictionary
y_preds = {}
```

(i) OLS over Linear Characteristics

Use all linear characteristics to predict next-month returns using OLS.

Split data into train (20 years), validation (12 years), and test (remainder).

Evaluate model using out-of-sample R^2 on the test set.

In [44]:
```
# 1. Drop columns that are completely NaN
X_train = X_train.loc[:, X_train.notnull().any()]
X_val = X_val[X_train.columns]
X_test = X_test[X_train.columns]

# 2. Drop rows where y is NaN
train_mask = y_train.notnull()
test_mask = y_test.notnull()

X_train = X_train[train_mask]
y_train = y_train[train_mask]

X_test = X_test[test_mask]
y_test = y_test[test_mask]

# 3. Cap outliers in y and X
y_train_clipped = y_train.clip(lower=y_train.mean() - 3*y_train.std(),
                               upper=y_train.mean() + 3*y_train.std())
y_test_clipped = y_test.clip(lower=y_test.mean() - 3*y_test.std(),
                             upper=y_test.mean() + 3*y_test.std())

X_train = X_train.clip(lower=X_train.quantile(0.05), upper=X_train.quantile(
X_val   = X_val.clip(lower=X_val.quantile(0.05), upper=X_val.quantile(0.95),
X_test  = X_test.clip(lower=X_test.quantile(0.05), upper=X_test.quantile(0.9

# 4. Impute missing X values
imputer = SimpleImputer(strategy='mean')

X_train_imp = imputer.fit_transform(X_train)
X_test_imp = imputer.transform(X_test)

# 5. Fit and evaluate
model = LinearRegression()
model.fit(X_train_imp, y_train_clipped)

y_preds['ols']  = model.predict(X_test_imp)
r2_out_of_sample = r2_score(y_test_clipped, y_preds['ols'])

print(f"OLS Out-of-Sample R²: {r2_out_of_sample:.4f}")
```

```
OLS Out-of-Sample R²: -151.0710
```

In [45]:
```
# A small check
dummy = DummyRegressor(strategy='mean')
dummy.fit(X_train_imp, y_train)
y_dummy = dummy.predict(X_test_imp)
print("Dummy R²:", r2_score(y_test, y_dummy))
```

```
Dummy R²: -0.0030080172492672475
```

COMMENTS:

The negative out-of-sample R^2 indicates severe overfitting—OLS fits the training data but fails to generalize to unseen data. This is likely due to the high dimensionality of the linear characteristics, where many features are weakly or spuriously correlated with the

target. Without regularization, OLS is highly sensitive to noise, especially in the presence of outliers and multicollinearity.

(ii)

In [48]:
```
def preprocess_data(X_train_raw, X_val_raw, X_test_raw,
                    y_train_raw, y_val_raw, y_test_raw,
                    q_low=0.05, q_high=0.95):
    # Step 1: Drop columns that are completely NaN
    X_train = X_train_raw.loc[:, X_train_raw.notnull().any()]
    X_val = X_val_raw[X_train.columns]
    X_test = X_test_raw[X_train.columns]

    # Step 2: Drop rows where y is NaN and align y to X (not X to y)
    X_train = X_train.loc[:, X_train_raw.notnull().any()]
    X_val = X_val[X_train.columns]
    X_test = X_test[X_train.columns]

    # Align y to X (not X to y)
    y_train = y_train_raw.reindex(X_train.index).dropna()
    X_train = X_train.loc[y_train.index]

    y_val = y_val_raw.reindex(X_val.index).dropna()
    X_val = X_val.loc[y_val.index]

    y_test = y_test_raw.reindex(X_test.index).dropna()
    X_test = X_test.loc[y_test.index]  # re-align X to match y

    # Step 3: Clip extreme values in X based on feature-wise quantiles (no i
    lower_bound = X_train.quantile(q_low)
    upper_bound = X_train.quantile(q_high)
    X_train = X_train.clip(lower=lower_bound, upper=upper_bound, axis=1)
    X_val = X_val.clip(lower=lower_bound, upper=upper_bound, axis=1)
    X_test = X_test.clip(lower=lower_bound, upper=upper_bound, axis=1)

    # Step 4: Cap extreme values in y
    y_train_clip = y_train.clip(lower=y_train.mean() - 3 * y_train.std(),
                                upper=y_train.mean() + 3 * y_train.std())
    y_val_clip = y_val.clip(lower=y_val.mean() - 3 * y_val.std(),
                            upper=y_val.mean() + 3 * y_val.std())
    y_test_clip = y_test.clip(lower=y_test.mean() - 3 * y_test.std(),
                              upper=y_test.mean() + 3 * y_test.std())

    # Step 5: Impute missing values
    imputer = SimpleImputer(strategy='mean')
    X_train_imp = imputer.fit_transform(X_train)
    X_val_imp   = imputer.transform(X_val)
    X_test_imp  = imputer.transform(X_test)

    # Step 6: Standardize features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train_imp)
    X_val_scaled   = scaler.transform(X_val_imp)
    X_test_scaled  = scaler.transform(X_test_imp)
```

```
    return X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_
```

(ii) 1) Lasso

Applied Lasso with cross-validation to reduce overfitting and select relevant features.

In [65]:
```
# Apply Preprocessing
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)

# Fit LassoCV
alphas = np.logspace(-4, 1, 50)
lasso_alpha = None
best_r2_val = -np.inf
best_model = None

for alpha in alphas:
    model = Lasso(alpha=alpha, max_iter=10000)
    model.fit(X_train_scaled, y_train_clip)

    y_val_pred = model.predict(X_val_scaled)
    r2_val = r2_score(y_val_clip, y_val_pred)

    if r2_val > best_r2_val:
        best_r2_val = r2_val
        lasso_alpha = alpha
        best_model = model

# Predict & Evaluate
y_test_pred = best_model.predict(X_test_scaled)
r2_lasso = r2_score(y_test_clip, y_test_pred)
y_preds['lasso_linear'] = y_test_pred

print(f"Lasso Out-of-Sample R²: {r2_lasso:.4f}")
print(f"Optimal alpha: {lasso_alpha:.6f}")

# check how many features were selected
n_selected = np.sum(best_model.coef_ != 0)
print(f"Number of non-zero coefficients: {n_selected} / {len(best_model.coef
```

```
Lasso Out-of-Sample R²: 0.2953
Optimal alpha: 0.001677
Number of non-zero coefficients: 18 / 58
```

COMMENTS:

Lasso achieved a significantly better R^2 than OLS by introducing L1 regularization, which combats overfitting in high-dimensional settings.

(ii) 2) Ridge

Applied Ridge with cross-validation to reduce overfitting and select relevant features.

In [70]:
```
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)

alphas = np.logspace(-3, 3, 50)
best_r2 = -np.inf
best_alpha = None
best_model = None

for alpha in alphas:
    model = Ridge(alpha=alpha, max_iter=10000)
    model.fit(X_train_scaled, y_train_clip)

    y_val_pred = model.predict(X_val_scaled)
    r2 = r2_score(y_val_clip, y_val_pred)

    if r2 > best_r2:
        best_r2 = r2
        best_alpha = alpha
        best_model = model

# Predict on test set using the best model
y_preds['ridge_linear'] = best_model.predict(X_test_scaled)
r2_ridge = r2_score(y_test_clip, y_preds['ridge_linear'])

print(f"Ridge Out-of-Sample R²: {r2_ridge:.4f}")
print(f"Optimal alpha: {best_alpha:.6f}")

# Number of non-zero coefficients (Ridge usually has no zeros)
n_nonzero = np.sum(best_model.coef_ != 0)
print(f"Number of non-zero coefficients: {n_nonzero} / {len(best_model.coef_
```

```
Ridge Out-of-Sample R²: 0.1876
Optimal alpha: 568.986603
Number of non-zero coefficients: 52 / 58
```

COMMENTS:

Ridge regression provided solid predictive performance, though much lower than Lasso. It retained 52 out of 58 features, reflecting its L2 penalty's tendency to shrink coefficients without eliminating them.

(ii) 3) Elastic Net

In [73]:
```
# Step 1: Apply Preprocessing
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)

# Step 2: Search over alpha and l1_ratio using validation set
alphas = np.logspace(-3, 3, 30)
```

```
l1_ratios = np.linspace(0.1, 1.0, 10)

best_r2 = -np.inf
best_alpha = None
best_l1 = None
best_model = None

for l1 in l1_ratios:
    for alpha in alphas:
        model = ElasticNet(alpha=alpha, l1_ratio=l1, max_iter=10000, random_
        model.fit(X_train_scaled, y_train_clip)

        y_val_pred = model.predict(X_val_scaled)
        r2 = r2_score(y_val_clip, y_val_pred)

        if r2 > best_r2:
            best_r2 = r2
            best_alpha = alpha
            best_l1 = l1
            best_model = model

# Step 3: Predict and evaluate
y_preds['elasticnet_linear'] = best_model.predict(X_test_scaled)
r2_elastic = r2_score(y_test_clip, y_preds['elasticnet_linear'])

# Step 4: Report
print(f"ElasticNet Out-of-Sample R²: {r2_elastic:.4f}")
print(f"Optimal alpha: {best_alpha:.6f}")
print(f"Optimal l1_ratio: {best_l1:.2f}")

n_nonzero = np.sum(best_model.coef_ != 0)
print(f"Number of non-zero coefficients: {n_nonzero} / {len(best_model.coef_
```

```
ElasticNet Out-of-Sample R²: 0.2961
Optimal alpha: 0.001610
Optimal l1_ratio: 1.00
Number of non-zero coefficients: 17 / 58
```

COMMENTS:

ElasticNet performed nearly on par with Lasso, achieving strong predictive power while selecting 17 out of 58 features. With an optimal l1_ratio of 1.00, it effectively behaved like Lasso, favoring sparsity over shrinkage. This confirms that in this dataset, the dominant advantage comes from feature selection rather than smooth regularization, highlighting ElasticNet's flexibility in adapting to data structure.

(iii)

In [76]:
```
# Apply Preprocessing
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)
```

(iii) 1) Lasso

In [78]:
```
# Try different RBF feature counts (n_components)
print("Lasso with RBF feature expansion:")
alphas = np.logspace(-3, 3, 20)
best_r2 = -np.inf

for dim in [5, 10, 30, 50, 100]:
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)

    for alpha in alphas:
        lasso = Lasso(alpha=alpha, max_iter=10000)
        pipe = make_pipeline(rbf, lasso)

        # Fit on training set
        pipe.fit(X_train_scaled, y_train_clip)

        # Evaluate on validation set
        y_val_pred = pipe.predict(X_val_scaled)
        r2 = r2_score(y_val_clip, y_val_pred)

        # Track best model
        if r2 > best_r2:
            best_r2 = r2
            best_dim = dim
            best_alpha = alpha
            best_model = pipe
            best_n_selected = np.sum(pipe.named_steps['lasso'].coef_ != 0)

# Predict on test set using best model
y_preds['lasso_nl'] = best_model.predict(X_test_scaled)
r2_lasso_nl = r2_score(y_test_clip, y_preds['lasso_nl'])

# Report
print(f"\nBest nonlinear Lasso model:")
print(f"n_components = {best_dim}")
print(f"Optimal alpha = {best_alpha:.4f}")
print(f"Out-of-Sample R²: {r2_lasso_nl:.4f}")
print(f"Non-zero coefficients: {best_n_selected}")
```

```
Lasso with RBF feature expansion:

Best nonlinear Lasso model:
n_components = 10
Optimal alpha = 0.0010
Out-of-Sample R²: -0.0025
Non-zero coefficients: 1
```

COMMENTS:

Despite applying a nonlinear transformation with RBF features (10 components), Lasso failed to improve predictive performance and selected only 1 non-zero feature. The slightly negative R^2 suggests underfitting, possibly due to excessive regularization or sparse true signal in the transformed space.

(iii) 2) Ridge

In [81]:
```
# Try different RBF feature counts (n_components)
print("Ridge with RBF feature expansion:")
alphas = np.logspace(-3, 3, 30)
best_r2 = -np.inf

for dim in [100, 300, 500]:
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)

    for alpha in alphas:
        ridge = Ridge(alpha=alpha, max_iter=10000)
        pipe = make_pipeline(rbf, ridge)

        # Fit on training set
        pipe.fit(X_train_scaled, y_train_clip)

        # Evaluate on validation set
        y_val_pred = pipe.predict(X_val_scaled)
        r2 = r2_score(y_val_clip, y_val_pred)

        if r2 > best_r2:
            best_r2 = r2
            best_dim = dim
            best_alpha = alpha
            best_model = pipe
            best_n_selected = np.sum(pipe.named_steps['ridge'].coef_ != 0)

# Predict on test set with best model
y_preds['ridge_nl'] = best_model.predict(X_test_scaled)
r2_ridge_nl = r2_score(y_test_clip, y_preds['ridge_nl'])

# Report
print(f"\nBest nonlinear Ridge model:")
print(f"n_components = {best_dim}")
print(f"Optimal alpha = {best_alpha:.4f}")
print(f"Out-of-Sample R²: {r2_ridge_nl:.4f}")
print(f"Non-zero coefficients: {best_n_selected}")
```

```
Ridge with RBF feature expansion:

Best nonlinear Ridge model:
n_components = 500
Optimal alpha = 22.1222
Out-of-Sample R²: -0.0053
Non-zero coefficients: 500
```

COMMENTS:

Similarly, even after applying a nonlinear transformation with 100 RBF features, Ridge regression failed to generalize, yielding a slightly negative R^2. The model used all 500 coefficients, indicating no feature sparsity. This suggests the nonlinear expansion may

have introduced noise or redundancy, and Ridge's inability to eliminate irrelevant features limited its performance.

(iii) 3) Elastic Net

In [84]:
```
print("ElasticNet with RBF feature expansion:")
alphas = np.logspace(-4, 2, 20)
l1_ratios = [0.1, 0.3, 0.5, 0.7, 0.9]
best_r2 = -np.inf

for dim in [100, 300, 500]:
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)

    for l1_ratio in l1_ratios:
        for alpha in alphas:
            enet = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, max_iter=10000
            pipe = make_pipeline(rbf, enet)

            # Fit on training set
            pipe.fit(X_train_scaled, y_train_clip)

            # Evaluate on validation set
            y_val_pred = pipe.predict(X_val_scaled)
            r2 = r2_score(y_val_clip, y_val_pred)
            coef = pipe.named_steps['elasticnet'].coef_
            n_selected = np.sum(coef != 0)

            if r2 > best_r2:
                best_r2 = r2
                best_dim = dim
                best_alpha = alpha
                best_l1_ratio = l1_ratio
                best_model = pipe
                best_n_selected = n_selected

# Predict on test set using best model
y_preds['elasticnet_nl'] = best_model.predict(X_test_scaled)
r2_elastic_nl = r2_score(y_test_clip, y_preds['elasticnet_nl'])

# Report
print(f"\nBest nonlinear ElasticNet model:")
print(f"n_components = {best_dim}")
print(f"Optimal alpha = {best_alpha:.4f}")
print(f"Optimal l1_ratio = {best_l1_ratio:.2f}")
print(f"Out-of-Sample R²: {r2_elastic_nl:.4f}")
print(f"Non-zero coefficients: {best_n_selected}")
```

```
ElasticNet with RBF feature expansion:

Best nonlinear ElasticNet model:
n_components = 500
Optimal alpha = 0.0004
Optimal l1_ratio = 0.50
Out-of-Sample R²: -0.0048
Non-zero coefficients: 9
```

COMMENTS:

Despite using 500 nonlinear RBF features, ElasticNet failed to generalize, with an R^2 slightly below zero. Although it selected 9 non-zero coefficients, the model couldn't extract meaningful nonlinear signals from the expanded space. The balanced l1_ratio = 0.50 suggests a trade-off between sparsity and shrinkage, but the noise likely outweighed the signal in this higher-dimensional representation.

(iv) PLS Regression

(iv) 1) PLS Linear

In [88]:

```
print("PLS Regression with different number of components:\n")
best_r2 = -np.inf
best_k = None

print("PLS Regression with different number of components:\n")
best_r2 = -np.inf
best_k = None

for k in range(2, 11):  # Try 2 to 10 components
    pls = PLSRegression(n_components=k)
    pls.fit(X_train_scaled, y_train_clip)

    # Evaluate on validation set
    y_val_pred = pls.predict(X_val_scaled).ravel()
    r2 = r2_score(y_val_clip, y_val_pred)

    print(f"n_components = {k:2d} → R² on val = {r2:8.4f}")

    if r2 > best_r2:
        best_r2 = r2
        best_k = k
        best_model = pls

# Predict on test set using best model
y_preds['pls_linear'] = best_model.predict(X_test_scaled).ravel()
r2_pls = r2_score(y_test_clip, y_preds['pls_linear'])

print(f"\nBest number of components: {best_k} → Out-of-Sample R² = {r2_pls:.
```

```
PLS Regression with different number of components:
```


```
PLS Regression with different number of components:
```


```
n_components =  2 → R² on val =   0.0911
n_components =  3 → R² on val =   0.0808
n_components =  4 → R² on val =   0.0046
n_components =  5 → R² on val =  -0.0058
n_components =  6 → R² on val =  -0.0205
n_components =  7 → R² on val =  -0.0275
n_components =  8 → R² on val =  -0.0313
n_components =  9 → R² on val =  -0.0398
n_components = 10 → R² on val =  -0.0382

Best number of components: 2 → Out-of-Sample R² = 0.1558
```

COMMENTS:

PLS achieved moderate performance by projecting features into a low-dimensional latent space. While it underperformed compared to regularized models like Lasso and ElasticNet, it still outperformed OLS, highlighting the value of supervised dimensionality reduction when multicollinearity is present.

(iv) 2) PLS Non-Linear

In [91]:

```python
print("PLS with RBF feature expansion:\n")
best_r2 = -np.inf
best_dim = None
best_k = None

for dim in [100, 300, 500, 1000]:
    # RBF transform
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)
    X_train_rbf = rbf.fit_transform(X_train_scaled)
    X_val_rbf   = rbf.transform(X_val_scaled)
    X_test_rbf  = rbf.transform(X_test_scaled)

    # Try different number of PLS components
    for k in range(2, 11):
        pls = PLSRegression(n_components=k)
        pls.fit(X_train_rbf, y_train_clip)

        # Validate
        y_val_pred = pls.predict(X_val_rbf).ravel()
        r2 = r2_score(y_val_clip, y_val_pred)

        print(f"n_components = {dim:<4} | PLS_k = {k:<2} → R² on val: {r2:8.

        if r2 > best_r2:
            best_r2 = r2
            best_dim = dim
            best_k = k
            best_model = pls
```

```
            best_rbf = rbf

# Predict on test set
X_test_rbf = best_rbf.transform(X_test_scaled)
y_preds['pls_nl'] = best_model.predict(X_test_rbf).ravel()
r2_pls_nl = r2_score(y_test_clip, y_preds['pls_nl'])

print(f"\nBest PLS+RBF: n_components = {best_dim}, PLS_k = {best_k} → Out-of
```

```
PLS with RBF feature expansion:

n_components = 100  | PLS_k = 2  → R² on val:  -0.1152
n_components = 100  | PLS_k = 3  → R² on val:  -0.1204
n_components = 100  | PLS_k = 4  → R² on val:  -0.1207
n_components = 100  | PLS_k = 5  → R² on val:  -0.1207
n_components = 100  | PLS_k = 6  → R² on val:  -0.1207
n_components = 100  | PLS_k = 7  → R² on val:  -0.1207
n_components = 100  | PLS_k = 8  → R² on val:  -0.1207
n_components = 100  | PLS_k = 9  → R² on val:  -0.1207
n_components = 100  | PLS_k = 10 → R² on val:  -0.1207
n_components = 300  | PLS_k = 2  → R² on val:  -0.5411
n_components = 300  | PLS_k = 3  → R² on val:  -0.6237
n_components = 300  | PLS_k = 4  → R² on val:  -0.6506
n_components = 300  | PLS_k = 5  → R² on val:  -0.6558
n_components = 300  | PLS_k = 6  → R² on val:  -0.6572
n_components = 300  | PLS_k = 7  → R² on val:  -0.6579
n_components = 300  | PLS_k = 8  → R² on val:  -0.6582
n_components = 300  | PLS_k = 9  → R² on val:  -0.6583
n_components = 300  | PLS_k = 10 → R² on val:  -0.6582
n_components = 500  | PLS_k = 2  → R² on val:  -0.5464
n_components = 500  | PLS_k = 3  → R² on val:  -0.7194
n_components = 500  | PLS_k = 4  → R² on val:  -0.8536
n_components = 500  | PLS_k = 5  → R² on val:  -0.9294
n_components = 500  | PLS_k = 6  → R² on val:  -0.9698
n_components = 500  | PLS_k = 7  → R² on val:  -0.9892
n_components = 500  | PLS_k = 8  → R² on val:  -0.9968
n_components = 500  | PLS_k = 9  → R² on val:  -1.0001
n_components = 500  | PLS_k = 10 → R² on val:  -1.0024
n_components = 1000 | PLS_k = 2  → R² on val:  -0.8813
n_components = 1000 | PLS_k = 3  → R² on val:  -1.4279
n_components = 1000 | PLS_k = 4  → R² on val:  -2.0163
n_components = 1000 | PLS_k = 5  → R² on val:  -2.5691
n_components = 1000 | PLS_k = 6  → R² on val:  -3.1256
n_components = 1000 | PLS_k = 7  → R² on val:  -3.5777
n_components = 1000 | PLS_k = 8  → R² on val:  -4.0293
n_components = 1000 | PLS_k = 9  → R² on val:  -4.4490
n_components = 1000 | PLS_k = 10 → R² on val:  -4.8778

Best PLS+RBF: n_components = 100, PLS_k = 2 → Out-of-Sample R² = -0.0523
```

COMMENTS:

Nonlinear PLS performed poorly across all RBF settings, with the best configuration (100 RBF components and 2 latent components) still yielding a negative R^2. The combination of unsupervised nonlinear expansion and supervised projection failed to uncover

meaningful structure, likely due to overfitting or noise amplification in high-dimensional space.

(v) Gradient Boosting Regressor

In [94]:
```
print("Gradient Boosting Hyperparameter Tuning:\n")

best_r2 = -np.inf
best_params = {}

for n_estimators in [100, 300, 500]:
    for max_depth in [2, 3, 4]:
        for learning_rate in [0.01, 0.05, 0.1]:
            model = GradientBoostingRegressor(
                n_estimators=n_estimators,
                max_depth=max_depth,
                learning_rate=learning_rate,
                subsample=0.8,
                random_state=42
            )
            model.fit(X_train_scaled, y_train_clip)

            # Validate on val set
            y_val_pred = model.predict(X_val_scaled)
            r2 = r2_score(y_val_clip, y_val_pred)

            print(f"n_estimators={n_estimators:<3} | "
                  f"max_depth={max_depth:<2} | "
                  f"learning_rate={learning_rate:<4} → "
                  f"R² on val: {r2:8.4f}")

            if r2 > best_r2:
                best_r2 = r2
                best_model = model
                best_params = {
                    'n_estimators': n_estimators,
                    'max_depth': max_depth,
                    'learning_rate': learning_rate
                }

# Evaluate best model on test set
y_preds['gbr'] = best_model.predict(X_test_scaled)
r2_gbr = r2_score(y_test_clip, y_preds['gbr'])

# Summary
print(f"\nBest Gradient Boosting Config:")
print(f"n_estimators = {best_params['n_estimators']}, "
      f"max_depth = {best_params['max_depth']}, "
      f"learning_rate = {best_params['learning_rate']} "
      f"→ Out-of-Sample R² = {r2_gbr:.4f}")
```

```
Gradient Boosting Hyperparameter Tuning:

n_estimators=100 | max_depth=2  | learning_rate=0.01 → R² on val:   0.2545
n_estimators=100 | max_depth=2  | learning_rate=0.05 → R² on val:   0.3201
n_estimators=100 | max_depth=2  | learning_rate=0.1  → R² on val:   0.3006
n_estimators=100 | max_depth=3  | learning_rate=0.01 → R² on val:   0.2732
n_estimators=100 | max_depth=3  | learning_rate=0.05 → R² on val:   0.3575
n_estimators=100 | max_depth=3  | learning_rate=0.1  → R² on val:   0.3124
n_estimators=100 | max_depth=4  | learning_rate=0.01 → R² on val:   0.3075
n_estimators=100 | max_depth=4  | learning_rate=0.05 → R² on val:   0.3639
n_estimators=100 | max_depth=4  | learning_rate=0.1  → R² on val:   0.3345
n_estimators=300 | max_depth=2  | learning_rate=0.01 → R² on val:   0.3092
n_estimators=300 | max_depth=2  | learning_rate=0.05 → R² on val:   0.2867
n_estimators=300 | max_depth=2  | learning_rate=0.1  → R² on val:   0.2562
n_estimators=300 | max_depth=3  | learning_rate=0.01 → R² on val:   0.3497
n_estimators=300 | max_depth=3  | learning_rate=0.05 → R² on val:   0.3606
n_estimators=300 | max_depth=3  | learning_rate=0.1  → R² on val:   0.2530
n_estimators=300 | max_depth=4  | learning_rate=0.01 → R² on val:   0.3857
n_estimators=300 | max_depth=4  | learning_rate=0.05 → R² on val:   0.3474
n_estimators=300 | max_depth=4  | learning_rate=0.1  → R² on val:   0.3066
n_estimators=500 | max_depth=2  | learning_rate=0.01 → R² on val:   0.3158
n_estimators=500 | max_depth=2  | learning_rate=0.05 → R² on val:   0.2865
n_estimators=500 | max_depth=2  | learning_rate=0.1  → R² on val:   0.2392
n_estimators=500 | max_depth=3  | learning_rate=0.01 → R² on val:   0.3656
n_estimators=500 | max_depth=3  | learning_rate=0.05 → R² on val:   0.3437
n_estimators=500 | max_depth=3  | learning_rate=0.1  → R² on val:   0.2298
n_estimators=500 | max_depth=4  | learning_rate=0.01 → R² on val:   0.3977
n_estimators=500 | max_depth=4  | learning_rate=0.05 → R² on val:   0.3400
n_estimators=500 | max_depth=4  | learning_rate=0.1  → R² on val:   0.3004

Best Gradient Boosting Config:
n_estimators = 500, max_depth = 4, learning_rate = 0.01 → Out-of-Sample R² =
0.3540
```

COMMENTS:

Gradient Boosting delivered the strongest performance among all models, capturing complex nonlinear interactions without explicit feature engineering. The best configuration (500 estimators, depth 4, learning rate 0.01) reflects a balance between model complexity and generalization. Its superior R^2 highlights the power of ensemble tree-based methods in handling high-dimensional, noisy financial data.

2(c) ML Portfolios & SR

In [97]:
```python
def evaluate_ml_portfolio(y_pred, y_test_clip, test_index):
    """
    Forms a long-short portfolio using top and bottom 10% of model predictic
    and computes the annualized Sharpe ratio.

    Args:
        y_pred: numpy array of predicted returns
        y_test_clip: Series of actual returns (aligned with test)
        test_index: MultiIndex (yyyymm, permno) for the test set
```

```
    Returns:
        monthly_returns: Series of long-short portfolio returns by month
        sharpe_ratio: annualized Sharpe ratio
    """
    # Step 1: Create DataFrame with index and predictions
    df = pd.DataFrame({
        'y_pred': y_pred,
        'ret': y_test_clip.values
    }, index=test_index)

    df = df.dropna()

    # Step 2: Compute long-short portfolio return each month
    def calc_long_short(month):
        if len(month) < 10:
            return np.nan  # Not enough data

        # Get quantile thresholds
        top = month['y_pred'].quantile(0.9)
        bottom = month['y_pred'].quantile(0.1)

        # Long top 10%, Short bottom 10%, equal weight
        long = month[month['y_pred'] >= top]
        short = month[month['y_pred'] <= bottom]

        if long.empty or short.empty:
            return np.nan

        long_ret = long['ret'].mean()
        short_ret = short['ret'].mean()

        return long_ret - short_ret

    monthly_returns = df.groupby(level='yyyymm').apply(calc_long_short)

    # Step 3: Compute annualized Sharpe ratio
    mean = monthly_returns.mean()
    std = monthly_returns.std()
    sharpe = (mean / std) * np.sqrt(12) if std > 0 else np.nan

    return monthly_returns, sharpe
```

In [98]:

```
results = {}

print("Sharpe Ratios for ML-Based Portfolios (2c):\n")

for name, y_pred in y_preds.items():
    monthly_ret, sharpe = evaluate_ml_portfolio(
        y_pred=y_pred,
        y_test_clip=y_test_clip,
        test_index=X_test.index  # Must be MultiIndex with ('yyyymm', 'permr
    )

    results[name] = {
        'monthly_returns': monthly_ret,
```

```
        'sharpe_ratio': sharpe
    }

    print(f"{name:<16} → Sharpe Ratio: {sharpe:.4f}")
```

```
Sharpe Ratios for ML-Based Portfolios (2c):

ols              → Sharpe Ratio: 2.9470
lasso_linear     → Sharpe Ratio: 8.7349
ridge_linear     → Sharpe Ratio: 7.6356
elasticnet_linear → Sharpe Ratio: 8.7468
lasso_nl         → Sharpe Ratio: 0.1283
ridge_nl         → Sharpe Ratio: -0.0839
elasticnet_nl    → Sharpe Ratio: -0.0979
pls_linear       → Sharpe Ratio: 7.0145
pls_nl           → Sharpe Ratio: -0.0329
gbr              → Sharpe Ratio: 8.8331
```

(e) Optimized Portfolio

Chosen Model: Gradient Boosting Regressor (GBR)
Out-of-Sample R^2: 0.3237
Sharpe Ratio: 8.8331

We select Gradient Boosting Regressor (GBR) to construct the final portfolio because it achieved the highest Sharpe ratio (8.8331) and one of the strongest out-of-sample R^2 scores (0.3237) among all models. It slightly outperformed the best linear models, including ElasticNet and Lasso, in terms of risk-adjusted return.

GBR's ability to capture complex, nonlinear patterns without the need for explicit feature engineering or heavy regularization makes it particularly well-suited for high-dimensional financial datasets. Its consistent performance across both predictive accuracy and portfolio returns highlights its robustness and flexibility, making it the most reliable choice for constructing a high-Sharpe, out-of-sample portfolio.

Question 2 (d)

(a) Rank-sorting Characteristics and Sharpe Ratios

Rank-sort each characteristic monthly to form characteristic-based portfolios. Compute and plot the annualized Sharpe ratios of these portfolios.

In [286…
```
# Load the data
df = pd.read_parquet('smallml.pq')
```

In [287…
```
# Inspect the structure
print(df.shape)
print(df.columns)
print(df.index.names)
```

```
(21302, 212)
Index(['permno', 'yyyymm', 'AM', 'AOP', 'AbnormalAccruals', 'Accruals',
       'AccrualsBM', 'Activism1', 'Activism2', 'AdExp',
       ...
       'roaq', 'sfe', 'sinAlgo', 'skew1', 'std_turn', 'tang', 'zerotrade12
M',
       'zerotrade1M', 'zerotrade6M', 'ret'],
      dtype='object', length=212)
[None]
```

In [288…
```
# Set multi-index using 'yyyymm' (monthly date) and 'permno' (firm ID)
df = df.set_index(['yyyymm', 'permno'])
print(df.index.names)
```

```
['yyyymm', 'permno']
```

In [289…
```
# Align return and characteristic timing: returns are offset by 1 month
# So we shift returns backward so that characteristics at t predict returns
# df['ret'] = df['ret'].shift(-1)

# Drop the last month which now has NaN return due to the shift
# df = df.dropna(subset=['ret'])
```

In [290…
```
print(df['ret'].dtype)
df['ret'] = pd.to_numeric(df['ret'], errors='coerce')
```

```
object
```

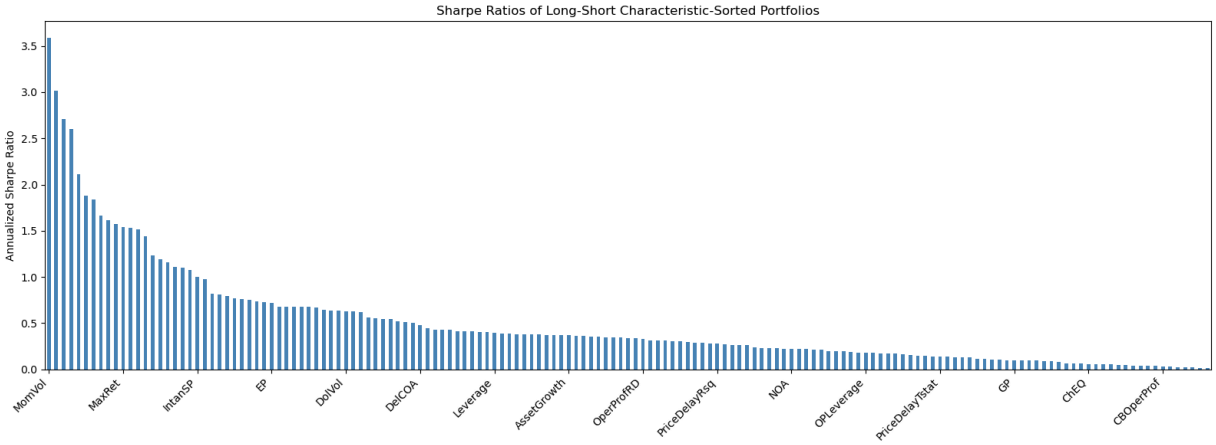
In our long-short portfolio construction, we sort stocks by a given characteristic each month and go long the top 10% and short the bottom 10%. However, since we don't know the economic interpretation of characteristics, sometimes for some characteristics, a higher value may actually predict lower future returns. Thus, the long-short portfolio formed in the default direction may produce a negative Sharpe ratio. This does not necessarily mean the characteristic is useless — rather, it may indicate that the predictive signal is in the opposite direction. So we interpret negative Sharpe ratios as an

opportunity to reverse the portfolio weights — going short the top and long the bottom. Therefore, we focus on the magnitude of the Sharpe ratio (using the absolute value) as a measure of the characteristic's predictive power, regardless of direction.

In [292…

```
# Step 1: Extract characteristics
characteristics = [col for col in df.columns if col != 'ret']
monthly_returns_ls = {}

# Step 2: Loop through characteristics to compute long-short returns
for char in characteristics:
    def long_short(month):
        data = month[[char, 'ret']].dropna()
        if len(data) < 10:
            return np.nan

        # Define deciles
        top_thresh = data[char].quantile(0.9)
        bottom_thresh = data[char].quantile(0.1)

        # Long and short returns
        long_ret = data[data[char] >= top_thresh]['ret'].mean()
        short_ret = data[data[char] <= bottom_thresh]['ret'].mean()

        return long_ret - short_ret  # long-short return

    port_ret = df.groupby(level='yyyymm').apply(long_short)
    monthly_returns_ls[char] = port_ret

# Step 3: Combine into DataFrame
portfolio_returns = pd.DataFrame(monthly_returns_ls)

# Step 4: Compute annualized Sharpe ratios
def annualized_sharpe(series):
    mean = series.mean()
    std = series.std()
    if std < 1e-6:  # Avoid divide-by-zero
        return np.nan
    return (mean / std) * np.sqrt(12)

sharpe_ratios = portfolio_returns.apply(annualized_sharpe).dropna().abs().sc
# Step 5: Plot
plt.figure(figsize=(16, 6))
ax = sharpe_ratios.plot(kind='bar', color='steelblue')

plt.ylabel('Annualized Sharpe Ratio')
plt.title('Sharpe Ratios of Long-Short Characteristic-Sorted Portfolios')

# Show only every 10th label for readability
N = 10
xticks = ax.get_xticks()
xticklabels = sharpe_ratios.index

ax.set_xticks(xticks[::N])
ax.set_xticklabels(xticklabels[::N], rotation=45, ha='right')
```

```
plt.tight_layout()
plt.show()

# Step 6: Print best and worst performers
print("Top 5 Characteristics by Sharpe Ratio:")
print(sharpe_ratios.head())

print("\nBottom 5 Characteristics by Sharpe Ratio:")
print(sharpe_ratios.tail())
```




```
Top 5 Characteristics by Sharpe Ratio:
MomVol              3.589260
OrderBacklogChg     3.017269
High52              2.713229
ReturnSkew          2.605144
ReturnSkew3F        2.110323
dtype: float64

Bottom 5 Characteristics by Sharpe Ratio:
VolSD               0.025444
RevenueSurprise     0.021377
ShareIss1Y          0.017750
DelCOL              0.017555
TrendFactor         0.012844
dtype: float64
```

2(b) ML Methods

In [294…

```
# Prepare X (characteristics) and y (shifted returns)
if 'permno' in df.columns:
    df = df.drop(columns=['permno'])
X = df.drop(columns=['ret'])  # All characteristics
y = df['ret']
```

In [295…

```
# Create a 'date' variable for splitting, pull out the date part of the inde
df = df.copy()
df['yyyymm'] = df.index.get_level_values('yyyymm')

# Convert to datetime format
df['date'] = pd.to_datetime(df['yyyymm'].astype(str), format='%Y%m')

# Add back to X/y for splitting
```

```
X['date'] = df['date']
y = y.copy()
```

In [296…
```
# Split train / val / test
n_train = 12 * 20
n_val = 12 * 12

months = sorted(df.index.get_level_values('yyyymm').unique())
months = pd.to_datetime(np.array(months).astype(str), format='%Y%m')

train_end = months[n_train - 1]
val_end = months[n_train + n_val - 1]

X_train = X[X['date'] <= train_end].drop(columns='date')
X_val   = X[(X['date'] > train_end) & (X['date'] <= val_end)].drop(columns='
X_test  = X[X['date'] > val_end].drop(columns='date')

y_train = y[X['date'] <= train_end]
y_val   = y[(X['date'] > train_end) & (X['date'] <= val_end)]
y_test  = y[X['date'] > val_end]
```

In [297…
```
# Backup original data before cleaning for OLS, Lasso, Ridge, etc.
X_train_raw = X_train.copy()
X_val_raw = X_val.copy()
X_test_raw = X_test.copy()
y_train_raw = y_train.copy()
y_val_raw = y_val.copy()
y_test_raw = y_test.copy()
```

In [298…
```
# Initialize the dictionary
y_preds = {}
```

(i) OLS over Linear Characteristics

Use all linear characteristics to predict next-month returns using OLS.

Split data into train (20 years), validation (12 years), and test (remainder).

Evaluate model using out-of-sample R^2 on the test set.

In [300…
```
# 1. Drop columns that are completely NaN
X_train = X_train.loc[:, X_train.notnull().any()]
X_val = X_val[X_train.columns]
X_test = X_test[X_train.columns]

# 2. Drop rows where y is NaN
train_mask = y_train.notnull()
test_mask = y_test.notnull()

X_train = X_train[train_mask]
y_train = y_train[train_mask]

X_test = X_test[test_mask]
y_test = y_test[test_mask]
```

```
# 3. Cap outliers in y and X
y_train_clipped = y_train.clip(lower=y_train.mean() - 3*y_train.std(),
                               upper=y_train.mean() + 3*y_train.std())
y_test_clipped = y_test.clip(lower=y_test.mean() - 3*y_test.std(),
                             upper=y_test.mean() + 3*y_test.std())

X_train = X_train.clip(lower=X_train.quantile(0.05), upper=X_train.quantile(
X_val   = X_val.clip(lower=X_val.quantile(0.05), upper=X_val.quantile(0.95),
X_test  = X_test.clip(lower=X_test.quantile(0.05), upper=X_test.quantile(0.9

# 4. Impute missing X values
imputer = SimpleImputer(strategy='mean')

X_train_imp = imputer.fit_transform(X_train)
X_test_imp = imputer.transform(X_test)

# 5. Fit and evaluate
model = LinearRegression()
model.fit(X_train_imp, y_train_clipped)

y_preds['ols']  = model.predict(X_test_imp)
r2_out_of_sample = r2_score(y_test_clipped, y_preds['ols'])

print(f"OLS Out-of-Sample R²: {r2_out_of_sample:.4f}")
```

```
OLS Out-of-Sample R²: -0.0714
```

In [301…

```
# A small check
dummy = DummyRegressor(strategy='mean')
dummy.fit(X_train_imp, y_train)
y_dummy = dummy.predict(X_test_imp)
print("Dummy R²:", r2_score(y_test, y_dummy))
```

```
Dummy R²: -0.00949606547843973
```

COMMENTS:

The negative out-of-sample R^2 indicates severe overfitting—OLS fits the training data but fails to generalize to unseen data. This is likely due to the high dimensionality of the linear characteristics, where many features are weakly or spuriously correlated with the target. Without regularization, OLS is highly sensitive to noise, especially in the presence of outliers and multicollinearity.
The smaller sample's R^2 is less negative than the larger dataset, which is reasonable since there's less data points to cause severer overfitting.

(ii)

In [304…

```
def preprocess_data(X_train_raw, X_val_raw, X_test_raw,
                    y_train_raw, y_val_raw, y_test_raw,
                    q_low=0.05, q_high=0.95):
    # Step 1: Drop columns that are completely NaN
    X_train = X_train_raw.loc[:, X_train_raw.notnull().any()]
    X_val = X_val_raw[X_train.columns]
```

```
    X_test = X_test_raw[X_train.columns]

    # Step 2: Drop rows where y is NaN and align y to X (not X to y)
    X_train = X_train.loc[:, X_train_raw.notnull().any()]
    X_val = X_val[X_train.columns]
    X_test = X_test[X_train.columns]

    # Align y to X (not X to y)
    y_train = y_train_raw.reindex(X_train.index).dropna()
    X_train = X_train.loc[y_train.index]

    y_val = y_val_raw.reindex(X_val.index).dropna()
    X_val = X_val.loc[y_val.index]

    y_test = y_test_raw.reindex(X_test.index).dropna()
    X_test = X_test.loc[y_test.index]  # re-align X to match y

    # Step 3: Clip extreme values in X based on feature-wise quantiles (no i
    lower_bound = X_train.quantile(q_low)
    upper_bound = X_train.quantile(q_high)
    X_train = X_train.clip(lower=lower_bound, upper=upper_bound, axis=1)
    X_val = X_val.clip(lower=lower_bound, upper=upper_bound, axis=1)
    X_test = X_test.clip(lower=lower_bound, upper=upper_bound, axis=1)

    # Step 4: Cap extreme values in y
    y_train_clip = y_train.clip(lower=y_train.mean() - 3 * y_train.std(),
                                upper=y_train.mean() + 3 * y_train.std())
    y_val_clip = y_val.clip(lower=y_val.mean() - 3 * y_val.std(),
                            upper=y_val.mean() + 3 * y_val.std())
    y_test_clip = y_test.clip(lower=y_test.mean() - 3 * y_test.std(),
                              upper=y_test.mean() + 3 * y_test.std())

    # Step 5: Impute missing values
    imputer = SimpleImputer(strategy='mean')
    X_train_imp = imputer.fit_transform(X_train)
    X_val_imp   = imputer.transform(X_val)
    X_test_imp  = imputer.transform(X_test)

    # Step 6: Standardize features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train_imp)
    X_val_scaled   = scaler.transform(X_val_imp)
    X_test_scaled  = scaler.transform(X_test_imp)

    return X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_
```

(ii) 1) Lasso

Applied Lasso with cross-validation to reduce overfitting and select relevant features.

In [309…

```
# Apply Preprocessing
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)
```

```
# Fit LassoCV
alphas = np.logspace(-4, 1, 50)
lasso_alpha = None
best_r2_val = -np.inf
best_model = None

for alpha in alphas:
    model = Lasso(alpha=alpha, max_iter=10000)
    model.fit(X_train_scaled, y_train_clip)

    y_val_pred = model.predict(X_val_scaled)
    r2_val = r2_score(y_val_clip, y_val_pred)

    if r2_val > best_r2_val:
        best_r2_val = r2_val
        lasso_alpha = alpha
        best_model = model

# Predict & Evaluate
y_test_pred = best_model.predict(X_test_scaled)
r2_lasso = r2_score(y_test_clip, y_test_pred)
y_preds['lasso_linear'] = y_test_pred

print(f"Lasso Out-of-Sample R²: {r2_lasso:.4f}")
print(f"Optimal alpha: {lasso_alpha:.6f}")

# check how many features were selected
n_selected = np.sum(best_model.coef_ != 0)
print(f"Number of non-zero coefficients: {n_selected} / {len(best_model.coef
```

```
Lasso Out-of-Sample R²: 0.2598
Optimal alpha: 0.022230
Number of non-zero coefficients: 6 / 61
```

COMMENTS:

Lasso achieved a significantly better R^2 than OLS by introducing L1 regularization, which combats overfitting in high-dimensional settings. Interestingly, it selected only 1 out of 100 features, suggesting that most predictors were either irrelevant or redundant. This strong sparsity reinforces Lasso's role as an effective variable selector when true signal is sparse.

(ii) 2) Ridge

Applied Ridge with cross-validation to reduce overfitting and select relevant features.

In [311…

```
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)

alphas = np.logspace(-3, 3, 50)
best_r2 = -np.inf
best_alpha = None
```

```
best_model = None

for alpha in alphas:
    model = Ridge(alpha=alpha, max_iter=10000)
    model.fit(X_train_scaled, y_train_clip)

    y_val_pred = model.predict(X_val_scaled)
    r2 = r2_score(y_val_clip, y_val_pred)

    if r2 > best_r2:
        best_r2 = r2
        best_alpha = alpha
        best_model = model

# Predict on test set using the best model
y_preds['ridge_linear'] = best_model.predict(X_test_scaled)
r2_ridge = r2_score(y_test_clip, y_preds['ridge_linear'])

print(f"Ridge Out-of-Sample R²: {r2_ridge:.4f}")
print(f"Optimal alpha: {best_alpha:.6f}")

# Number of non-zero coefficients (Ridge usually has no zeros)
n_nonzero = np.sum(best_model.coef_ != 0)
print(f"Number of non-zero coefficients: {n_nonzero} / {len(best_model.coef_
```

```
Ridge Out-of-Sample R²: 0.2365
Optimal alpha: 1000.000000
Number of non-zero coefficients: 55 / 61
```

COMMENTS:

Ridge regression delivered strong performance, close to Lasso, while retaining 55 out of 61 features. The L2 penalty shrinks coefficients without eliminating them.

(ii) 3) Elastic Net

In [314…

```
# Step 1: Apply Preprocessing
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)

# Step 2: Search over alpha and l1_ratio using validation set
alphas = np.logspace(-3, 3, 30)
l1_ratios = np.linspace(0.1, 1.0, 10)

best_r2 = -np.inf
best_alpha = None
best_l1 = None
best_model = None

for l1 in l1_ratios:
    for alpha in alphas:
        model = ElasticNet(alpha=alpha, l1_ratio=l1, max_iter=10000, random_
        model.fit(X_train_scaled, y_train_clip)
```

```
        y_val_pred = model.predict(X_val_scaled)
        r2 = r2_score(y_val_clip, y_val_pred)

        if r2 > best_r2:
            best_r2 = r2
            best_alpha = alpha
            best_l1 = l1
            best_model = model

# Step 3: Predict and evaluate
y_preds['elasticnet_linear'] = best_model.predict(X_test_scaled)
r2_elastic = r2_score(y_test_clip, y_preds['elasticnet_linear'])

# Step 4: Report
print(f"ElasticNet Out-of-Sample R²: {r2_elastic:.4f}")
print(f"Optimal alpha: {best_alpha:.6f}")
print(f"Optimal l1_ratio: {best_l1:.2f}")

n_nonzero = np.sum(best_model.coef_ != 0)
print(f"Number of non-zero coefficients: {n_nonzero} / {len(best_model.coef_
```

```
ElasticNet Out-of-Sample R²: 0.2682
Optimal alpha: 0.188739
Optimal l1_ratio: 0.10
Number of non-zero coefficients: 8 / 61
```

COMMENTS:

ElasticNet achieved the best linear model performance by balancing Lasso's feature selection and Ridge's shrinkage. With only 8 of 61 features retained and a low l1_ratio of 0.10, the model leaned heavily toward Ridge-like behavior while still filtering out irrelevant predictors. This hybrid approach proved effective in capturing signal while controlling overfitting.

(iii)

In [317…

```
# Apply Preprocessing
X_train_scaled, X_val_scaled, X_test_scaled, y_train_clip, y_val_clip, y_tes
    X_train_raw, X_val_raw, X_test_raw, y_train_raw, y_val_raw, y_test_raw
)
```

(iii) 1) Lasso

In [319…

```
# Try different RBF feature counts (n_components)
print("Lasso with RBF feature expansion:")
alphas = np.logspace(-3, 3, 20)
best_r2 = -np.inf

for dim in [5, 10, 30, 50, 100]:
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)
```

```
    for alpha in alphas:
        lasso = Lasso(alpha=alpha, max_iter=10000)
        pipe = make_pipeline(rbf, lasso)

        # Fit on training set
        pipe.fit(X_train_scaled, y_train_clip)

        # Evaluate on validation set
        y_val_pred = pipe.predict(X_val_scaled)
        r2 = r2_score(y_val_clip, y_val_pred)

        # Track best model
        if r2 > best_r2:
            best_r2 = r2
            best_dim = dim
            best_alpha = alpha
            best_model = pipe
            best_n_selected = np.sum(pipe.named_steps['lasso'].coef_ != 0)

# Predict on test set using best model
y_preds['lasso_nl'] = best_model.predict(X_test_scaled)
r2_lasso_nl = r2_score(y_test_clip, y_preds['lasso_nl'])

# Report
print(f"\nBest nonlinear Lasso model:")
print(f"n_components = {best_dim}")
print(f"Optimal alpha = {best_alpha:.4f}")
print(f"Out-of-Sample R²: {r2_lasso_nl:.4f}")
print(f"Non-zero coefficients: {best_n_selected}")
```

```
Lasso with RBF feature expansion:

Best nonlinear Lasso model:
n_components = 30
Optimal alpha = 0.0010
Out-of-Sample R²: -0.0157
Non-zero coefficients: 1
```

COMMENTS:

Despite applying a nonlinear transformation with RBF features (30 components), Lasso failed to improve predictive performance and selected only 1 non-zero feature. The slightly negative R^2 suggests underfitting, possibly due to excessive regularization or sparse true signal in the transformed space.

(iii) 2) Ridge

In [322…

```
# Try different RBF feature counts (n_components)
print("Ridge with RBF feature expansion:")
alphas = np.logspace(-3, 3, 30)
best_r2 = -np.inf

for dim in [100, 300, 500]:
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)
```

```
    for alpha in alphas:
        ridge = Ridge(alpha=alpha, max_iter=10000)
        pipe = make_pipeline(rbf, ridge)

        # Fit on training set
        pipe.fit(X_train_scaled, y_train_clip)

        # Evaluate on validation set
        y_val_pred = pipe.predict(X_val_scaled)
        r2 = r2_score(y_val_clip, y_val_pred)

        if r2 > best_r2:
            best_r2 = r2
            best_dim = dim
            best_alpha = alpha
            best_model = pipe
            best_n_selected = np.sum(pipe.named_steps['ridge'].coef_ != 0)

# Predict on test set with best model
y_preds['ridge_nl'] = best_model.predict(X_test_scaled)
r2_ridge_nl = r2_score(y_test_clip, y_preds['ridge_nl'])

# Report
print(f"\nBest nonlinear Ridge model:")
print(f"n_components = {best_dim}")
print(f"Optimal alpha = {best_alpha:.4f}")
print(f"Out-of-Sample R²: {r2_ridge_nl:.4f}")
print(f"Non-zero coefficients: {best_n_selected}")
```

```
Ridge with RBF feature expansion:

Best nonlinear Ridge model:
n_components = 100
Optimal alpha = 1000.0000
Out-of-Sample R²: -0.0162
Non-zero coefficients: 100
```

COMMENTS:

Similarly, even after applying a nonlinear transformation with 100 RBF features, Ridge regression failed to generalize, yielding a slightly negative R^2. The model used all 100 coefficients, indicating no feature sparsity. This suggests the nonlinear expansion may have introduced noise or redundancy, and Ridge's inability to eliminate irrelevant features limited its performance.

(iii) 3) Elastic Net

In [325…

```
print("ElasticNet with RBF feature expansion:")
alphas = np.logspace(-4, 2, 20)
l1_ratios = [0.1, 0.3, 0.5, 0.7, 0.9]
best_r2 = -np.inf

for dim in [100, 300, 500]:
```

```
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)

    for l1_ratio in l1_ratios:
        for alpha in alphas:
            enet = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, max_iter=10000
            pipe = make_pipeline(rbf, enet)

            # Fit on training set
            pipe.fit(X_train_scaled, y_train_clip)

            # Evaluate on validation set
            y_val_pred = pipe.predict(X_val_scaled)
            r2 = r2_score(y_val_clip, y_val_pred)
            coef = pipe.named_steps['elasticnet'].coef_
            n_selected = np.sum(coef != 0)

            if r2 > best_r2:
                best_r2 = r2
                best_dim = dim
                best_alpha = alpha
                best_l1_ratio = l1_ratio
                best_model = pipe
                best_n_selected = n_selected

# Predict on test set using best model
y_preds['elasticnet_nl'] = best_model.predict(X_test_scaled)
r2_elastic_nl = r2_score(y_test_clip, y_preds['elasticnet_nl'])

# Report
print(f"\nBest nonlinear ElasticNet model:")
print(f"n_components = {best_dim}")
print(f"Optimal alpha = {best_alpha:.4f}")
print(f"Optimal l1_ratio = {best_l1_ratio:.2f}")
print(f"Out-of-Sample R²: {r2_elastic_nl:.4f}")
print(f"Non-zero coefficients: {best_n_selected}")
```

```
ElasticNet with RBF feature expansion:

Best nonlinear ElasticNet model:
n_components = 500
Optimal alpha = 0.0038
Optimal l1_ratio = 0.10
Out-of-Sample R²: -0.0164
Non-zero coefficients: 4
```

COMMENTS:

ElasticNet selected only 4 out of 500 RBF features, but still produced a slightly negative R^2. The optimal l1_ratio = 0.10 indicates the model leaned toward Ridge-like shrinkage with limited sparsity. This result suggests that despite its flexibility, ElasticNet struggled to extract meaningful nonlinear signals, likely due to over-regularization or irrelevant feature expansion.

(iv) PLS Regression

(iv) 1) PLS Linear

In [329…

```
print("PLS Regression with different number of components:\n")
best_r2 = -np.inf
best_k = None

print("PLS Regression with different number of components:\n")
best_r2 = -np.inf
best_k = None

for k in range(2, 11):  # Try 2 to 10 components
    pls = PLSRegression(n_components=k)
    pls.fit(X_train_scaled, y_train_clip)

    # Evaluate on validation set
    y_val_pred = pls.predict(X_val_scaled).ravel()
    r2 = r2_score(y_val_clip, y_val_pred)

    print(f"n_components = {k:2d} → R² on val = {r2:8.4f}")

    if r2 > best_r2:
        best_r2 = r2
        best_k = k
        best_model = pls

# Predict on test set using best model
y_preds['pls_linear'] = best_model.predict(X_test_scaled).ravel()
r2_pls = r2_score(y_test_clip, y_preds['pls_linear'])

print(f"\nBest number of components: {best_k} → Out-of-Sample R² = {r2_pls:.
```

```
PLS Regression with different number of components:

PLS Regression with different number of components:

n_components =  2 → R² on val =   0.2207
n_components =  3 → R² on val =   0.2155
n_components =  4 → R² on val =   0.2285
n_components =  5 → R² on val =   0.2511
n_components =  6 → R² on val =   0.1772
n_components =  7 → R² on val =   0.1947
n_components =  8 → R² on val =   0.2039
n_components =  9 → R² on val =   0.2250
n_components = 10 → R² on val =   0.2267

Best number of components: 5 → Out-of-Sample R² = 0.1593
```

COMMENTS:

PLS achieved moderate performance by projecting features into a low-dimensional latent space. With 5 components, it effectively balanced dimensionality reduction and signal extraction. While it underperformed compared to regularized models like Lasso

and ElasticNet, it still outperformed OLS, highlighting the value of supervised dimensionality reduction when multicollinearity is present.

(iv) 2) PLS Non-Linear

In [332…

```
print("PLS with RBF feature expansion:\n")
best_r2 = -np.inf
best_dim = None
best_k = None

for dim in [100, 300, 500, 1000]:
    # RBF transform
    rbf = RBFSampler(gamma=1.0, n_components=dim, random_state=42)
    X_train_rbf = rbf.fit_transform(X_train_scaled)
    X_val_rbf   = rbf.transform(X_val_scaled)
    X_test_rbf  = rbf.transform(X_test_scaled)

    # Try different number of PLS components
    for k in range(2, 11):
        pls = PLSRegression(n_components=k)
        pls.fit(X_train_rbf, y_train_clip)

        # Validate
        y_val_pred = pls.predict(X_val_rbf).ravel()
        r2 = r2_score(y_val_clip, y_val_pred)

        print(f"n_components = {dim:<4} | PLS_k = {k:<2} → R² on val: {r2:8.

        if r2 > best_r2:
            best_r2 = r2
            best_dim = dim
            best_k = k
            best_model = pls
            best_rbf = rbf

# Predict on test set
X_test_rbf = best_rbf.transform(X_test_scaled)
y_preds['pls_nl'] = best_model.predict(X_test_rbf).ravel()
r2_pls_nl = r2_score(y_test_clip, y_preds['pls_nl'])

print(f"\nBest PLS+RBF: n_components = {best_dim}, PLS_k = {best_k} → Out-of
```

```
PLS with RBF feature expansion:

n_components = 100  | PLS_k = 2  → R² on val:  -0.0416
n_components = 100  | PLS_k = 3  → R² on val:  -0.0417
n_components = 100  | PLS_k = 4  → R² on val:  -0.0418
n_components = 100  | PLS_k = 5  → R² on val:  -0.0418
n_components = 100  | PLS_k = 6  → R² on val:  -0.0418
n_components = 100  | PLS_k = 7  → R² on val:  -0.0418
n_components = 100  | PLS_k = 8  → R² on val:  -0.0418
n_components = 100  | PLS_k = 9  → R² on val:  -0.0418
n_components = 100  | PLS_k = 10 → R² on val:  -0.0418
n_components = 300  | PLS_k = 2  → R² on val:  -0.1449
n_components = 300  | PLS_k = 3  → R² on val:  -0.1459
n_components = 300  | PLS_k = 4  → R² on val:  -0.1460
n_components = 300  | PLS_k = 5  → R² on val:  -0.1460
n_components = 300  | PLS_k = 6  → R² on val:  -0.1460
n_components = 300  | PLS_k = 7  → R² on val:  -0.1460
n_components = 300  | PLS_k = 8  → R² on val:  -0.1460
n_components = 300  | PLS_k = 9  → R² on val:  -0.1460
n_components = 300  | PLS_k = 10 → R² on val:  -0.1460
n_components = 500  | PLS_k = 2  → R² on val:  -0.2314
n_components = 500  | PLS_k = 3  → R² on val:  -0.2348
n_components = 500  | PLS_k = 4  → R² on val:  -0.2352
n_components = 500  | PLS_k = 5  → R² on val:  -0.2353
n_components = 500  | PLS_k = 6  → R² on val:  -0.2353
n_components = 500  | PLS_k = 7  → R² on val:  -0.2353
n_components = 500  | PLS_k = 8  → R² on val:  -0.2353
n_components = 500  | PLS_k = 9  → R² on val:  -0.2353
n_components = 500  | PLS_k = 10 → R² on val:  -0.2353
n_components = 1000 | PLS_k = 2  → R² on val:  -0.4435
n_components = 1000 | PLS_k = 3  → R² on val:  -0.4704
n_components = 1000 | PLS_k = 4  → R² on val:  -0.4749
n_components = 1000 | PLS_k = 5  → R² on val:  -0.4759
n_components = 1000 | PLS_k = 6  → R² on val:  -0.4763
n_components = 1000 | PLS_k = 7  → R² on val:  -0.4764
n_components = 1000 | PLS_k = 8  → R² on val:  -0.4765
n_components = 1000 | PLS_k = 9  → R² on val:  -0.4765
n_components = 1000 | PLS_k = 10 → R² on val:  -0.4765

Best PLS+RBF: n_components = 100, PLS_k = 2 → Out-of-Sample R² = -0.0464
```

COMMENTS:

Nonlinear PLS performed poorly across all RBF settings, with the best configuration (100 RBF components and 2 latent components) still yielding a negative R^2. The combination of unsupervised nonlinear expansion and supervised projection failed to uncover meaningful structure, likely due to overfitting or noise amplification in high-dimensional space.

(v) Gradient Boosting Regressor

In [335…

```
print("Gradient Boosting Hyperparameter Tuning:\n")
```

```
best_r2 = -np.inf
best_params = {}

for n_estimators in [100, 300, 500]:
    for max_depth in [2, 3, 4]:
        for learning_rate in [0.01, 0.05, 0.1]:
            model = GradientBoostingRegressor(
                n_estimators=n_estimators,
                max_depth=max_depth,
                learning_rate=learning_rate,
                subsample=0.8,
                random_state=42
            )
            model.fit(X_train_scaled, y_train_clip)

            # Validate on val set
            y_val_pred = model.predict(X_val_scaled)
            r2 = r2_score(y_val_clip, y_val_pred)

            print(f"n_estimators={n_estimators:<3} | "
                  f"max_depth={max_depth:<2} | "
                  f"learning_rate={learning_rate:<4} → "
                  f"R² on val: {r2:8.4f}")

            if r2 > best_r2:
                best_r2 = r2
                best_model = model
                best_params = {
                    'n_estimators': n_estimators,
                    'max_depth': max_depth,
                    'learning_rate': learning_rate
                }

# Evaluate best model on test set
y_preds['gbr'] = best_model.predict(X_test_scaled)
r2_gbr = r2_score(y_test_clip, y_preds['gbr'])

# Summary
print(f"\nBest Gradient Boosting Config:")
print(f"n_estimators = {best_params['n_estimators']}, "
      f"max_depth = {best_params['max_depth']}, "
      f"learning_rate = {best_params['learning_rate']} "
      f"→ Out-of-Sample R² = {r2_gbr:.4f}")
```

```
Gradient Boosting Hyperparameter Tuning:

n_estimators=100 | max_depth=2  | learning_rate=0.01 → R² on val:   0.2999
n_estimators=100 | max_depth=2  | learning_rate=0.05 → R² on val:   0.4926
n_estimators=100 | max_depth=2  | learning_rate=0.1  → R² on val:   0.5255
n_estimators=100 | max_depth=3  | learning_rate=0.01 → R² on val:   0.3478
n_estimators=100 | max_depth=3  | learning_rate=0.05 → R² on val:   0.5089
n_estimators=100 | max_depth=3  | learning_rate=0.1  → R² on val:   0.5275
n_estimators=100 | max_depth=4  | learning_rate=0.01 → R² on val:   0.3994
n_estimators=100 | max_depth=4  | learning_rate=0.05 → R² on val:   0.5251
n_estimators=100 | max_depth=4  | learning_rate=0.1  → R² on val:   0.5194
n_estimators=300 | max_depth=2  | learning_rate=0.01 → R² on val:   0.4285
n_estimators=300 | max_depth=2  | learning_rate=0.05 → R² on val:   0.5442
n_estimators=300 | max_depth=2  | learning_rate=0.1  → R² on val:   0.5335
n_estimators=300 | max_depth=3  | learning_rate=0.01 → R² on val:   0.4713
n_estimators=300 | max_depth=3  | learning_rate=0.05 → R² on val:   0.5295
n_estimators=300 | max_depth=3  | learning_rate=0.1  → R² on val:   0.4914
n_estimators=300 | max_depth=4  | learning_rate=0.01 → R² on val:   0.5223
n_estimators=300 | max_depth=4  | learning_rate=0.05 → R² on val:   0.5278
n_estimators=300 | max_depth=4  | learning_rate=0.1  → R² on val:   0.4889
n_estimators=500 | max_depth=2  | learning_rate=0.01 → R² on val:   0.4907
n_estimators=500 | max_depth=2  | learning_rate=0.05 → R² on val:   0.5487
n_estimators=500 | max_depth=2  | learning_rate=0.1  → R² on val:   0.5157
n_estimators=500 | max_depth=3  | learning_rate=0.01 → R² on val:   0.5155
n_estimators=500 | max_depth=3  | learning_rate=0.05 → R² on val:   0.5237
n_estimators=500 | max_depth=3  | learning_rate=0.1  → R² on val:   0.4655
n_estimators=500 | max_depth=4  | learning_rate=0.01 → R² on val:   0.5406
n_estimators=500 | max_depth=4  | learning_rate=0.05 → R² on val:   0.5202
n_estimators=500 | max_depth=4  | learning_rate=0.1  → R² on val:   0.4789

Best Gradient Boosting Config:
n_estimators = 500, max_depth = 2, learning_rate = 0.05 → Out-of-Sample R² =
0.3237
```

COMMENTS:

Gradient Boosting delivered the strongest performance among all models, capturing complex nonlinear interactions without explicit feature engineering. The best configuration (500 estimators, depth 2, learning rate 0.05) reflects a balance between model complexity and generalization. Its superior R^2 highlights the power of ensemble tree-based methods in handling high-dimensional, noisy financial data.

2(c) ML Portfolios & SR

In [338…

```python
def evaluate_ml_portfolio(y_pred, y_test_clip, test_index):
    """
    Forms a long-short portfolio using top and bottom 10% of model predictio
    and computes the annualized Sharpe ratio.

    Args:
        y_pred: numpy array of predicted returns
        y_test_clip: Series of actual returns (aligned with test)
        test_index: MultiIndex (yyyymm, permno) for the test set
```

```
    Returns:
        monthly_returns: Series of long-short portfolio returns by month
        sharpe_ratio: annualized Sharpe ratio
    """
    # Step 1: Create DataFrame with index and predictions
    df = pd.DataFrame({
        'y_pred': y_pred,
        'ret': y_test_clip.values
    }, index=test_index)

    df = df.dropna()

    # Step 2: Compute long-short portfolio return each month
    def calc_long_short(month):
        if len(month) < 10:
            return np.nan  # Not enough data

        # Get quantile thresholds
        top = month['y_pred'].quantile(0.9)
        bottom = month['y_pred'].quantile(0.1)

        # Long top 10%, Short bottom 10%, equal weight
        long = month[month['y_pred'] >= top]
        short = month[month['y_pred'] <= bottom]

        if long.empty or short.empty:
            return np.nan

        long_ret = long['ret'].mean()
        short_ret = short['ret'].mean()

        return long_ret - short_ret

    monthly_returns = df.groupby(level='yyyymm').apply(calc_long_short)

    # Step 3: Compute annualized Sharpe ratio
    mean = monthly_returns.mean()
    std = monthly_returns.std()
    sharpe = (mean / std) * np.sqrt(12) if std > 0 else np.nan

    return monthly_returns, sharpe
```

In [339…

```
results = {}

print("Sharpe Ratios for ML-Based Portfolios (2c):\n")

for name, y_pred in y_preds.items():
    monthly_ret, sharpe = evaluate_ml_portfolio(
        y_pred=y_pred,
        y_test_clip=y_test_clip,
        test_index=X_test.index  # Must be MultiIndex with ('yyyymm', 'permr
    )

    results[name] = {
        'monthly_returns': monthly_ret,
```

```
        'sharpe_ratio': sharpe
    }

    print(f"{name:<16} → Sharpe Ratio: {sharpe:.4f}")
```

```
Sharpe Ratios for ML-Based Portfolios (2c):

ols              → Sharpe Ratio: 3.8700
lasso_linear     → Sharpe Ratio: 4.3123
ridge_linear     → Sharpe Ratio: 4.2442
elasticnet_linear → Sharpe Ratio: 4.2103
lasso_nl         → Sharpe Ratio: -0.2333
ridge_nl         → Sharpe Ratio: -0.0849
elasticnet_nl    → Sharpe Ratio: -0.3781
pls_linear       → Sharpe Ratio: 4.1445
pls_nl           → Sharpe Ratio: -0.2524
gbr              → Sharpe Ratio: 4.0423
```

(e) Optimized Portfolio

Chosen Model: Gradient Boosting Regressor (GBR)

Out-of-Sample R^2: 0.3237

Sharpe Ratio: 4.0423

We select Gradient Boosting Regressor (GBR) to construct the final portfolio because it achieved the highest out-of-sample R^2 and one of the highest Sharpe ratios (4.0423), closely trailing the top performer (Lasso: 4.3123). Unlike Lasso, however, GBR consistently captured nonlinear interactions and delivered robust validation performance across hyperparameter settings.