University of Chicago
Booth School of Business
Homework 1

Machine Learning In Finance - BUSN 35137

Frank Wang

Helen Du

Trenton Potter

Question 1

Imports & Preprocessing

In [68]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from collections import defaultdict
from typing import Tuple, Optional, Dict, List

# Scikit-Learn imports
from sklearn.linear_model import (
    RidgeCV, LassoCV, ElasticNetCV, LinearRegression, Ridge, Lasso, ElasticNet
)
from sklearn.metrics import r2_score
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.kernel_approximation import RBFSampler
from sklearn.kernel_ridge import KernelRidge
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.pipeline import make_pipeline
from sklearn.base import BaseEstimator

# Parallel processing imports
from joblib import Parallel, delayed
from tqdm import tqdm
from tqdm_joblib import tqdm_joblib
```

In [69]:
```
# Load the CSV file into a pandas DataFrame

dir = "/home/tpot/Code/W24/MLF/HW1/"
df = pd.read_csv(dir + 'gw.csv')

df['yyyymm'] = pd.to_datetime(df['yyyymm'], format='%Y%m', errors='coerce')
df.set_index('yyyymm', inplace=True)
df.rename(columns={'CRSP_SPvw_minus_Rfree': 're'}, inplace=True)
# Regularize all columns of the DataFrame, this is technically sloppy and introduces leakage
df = (df - df.mean()) / df.std()
df
```

Out[69]:

yyyymm	dfy_lag1	infl_lag1	svar_lag1	de_lag1	lty_lag1	tms_lag1	tbl_lag1	dfr_lag1	dp_lag1	dy_lag1	ltr_lag1	ep_lag1	b/m_lag1	ntis_lag1
1927-01-01	-0.178567	-0.455430	-0.404446	0.164764	-0.530471	-0.931014	-0.089500	-0.181572	0.897487	0.926446	0.118242	0.870843	-0.437550	1.339036
1927-02-01	-0.251722	-2.599994	-0.403574	0.221762	-0.541174	-1.076961	-0.037397	-0.160243	0.963220	0.911832	0.106069	0.899655	-0.429207	1.337010
1927-03-01	-0.295614	-1.539967	-0.433733	0.278285	-0.555445	-1.153775	-0.017858	-0.160243	0.883493	0.977369	0.158817	0.766913	-0.486100	1.369631
1927-04-01	-0.295614	-1.546200	-0.328524	0.332680	-0.612528	-1.207545	-0.047166	-1.233798	0.889929	0.897381	0.828303	0.731743	-0.331698	1.164322
1927-05-01	-0.324876	-0.455430	-0.381653	0.389574	-0.605392	-1.338128	0.014706	0.401419	0.873430	0.903655	-0.218530	0.669098	-0.380382	1.325036
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2020-08-01	0.070158	0.504594	-0.215570	0.399609	-1.572234	-0.915651	-1.046887	2.271253	-1.324108	-1.225455	0.966257	-1.784995	-1.139615	-1.091495
2020-09-01	-0.149305	0.143030	-0.358388	0.395711	-1.561531	-0.869563	-1.056656	-1.013399	-1.479440	-1.351164	-1.614307	-1.954875	-1.206492	-0.956417
2020-10-01	-0.105413	-0.191095	0.329506	0.391782	-1.550828	-0.854200	-1.053400	-0.302436	-1.403671	-1.507275	0.126357	-1.867471	-1.185886	-0.847953
2020-11-01	-0.046889	-0.376677	0.123624	0.424710	-1.511584	-0.762023	-1.056656	0.316103	-1.350445	-1.428012	-1.163925	-1.833848	-1.142241	-0.700942
2020-12-01	-0.178567	-0.571323	-0.069521	0.458214	-1.483042	-0.692891	-1.059913	2.932449	-1.576587	-1.374558	0.179104	-2.111663	-1.242498	-0.831088

1128 rows × 15 columns

Question 1a

We regress S&P 500 excess returns r_t on each predictor $x_t^{(j)}$ individually using:

$$r_t = \alpha + \beta_j x_t^{(j)} + \varepsilon_t$$

- **In-sample** R^2 is computed using the full dataset.
- **Out-of-sample** R^2 is computed recursively via an expanding window starting in 1965:

$$R^2_{\text{oos},t} = 1 - \frac{\sum_{s=1}^{T}(r_s - \hat{r}_s)^2}{\sum_{s=1}^{T}(r_s - \bar{r})^2}$$

Where:

- $\hat{r}_s$ is the model's prediction using parameters estimated up to time $s-1$
- $\bar{r}$ is the mean of historical returns in the full sample

Observed Results

- **In-sample** R^2 values for predictors were small and positive:
  $R^2_{\text{in-sample}} \in [0.005, 0.0001]$
- **Out-of-sample** R^2 values were uniformly negative

Interpretation

- The small in-sample R^2 values suggest that only a tiny fraction of the variance in market returns is explained by these predictors in-sample.
- The negative out-of-sample R^2 indicates that predictive performance worsens out-of-sample compared to just predicting the mean.

These results suggest that the predictors, while appearing weakly correlated with returns in-sample, do not generalize and offer no robust out-of-sample predictive power for S&P 500 excess returns.

This highlights a classic problem in return forecasting:

- Spurious in-sample fits due to noise
- Lack of economic signal or stability in the predictor-return relationship
- Potential overfitting when not penalizing or regularizing

In [70]:
```
## Params

start_year = pd.Timestamp(year=1965, month=1, day=1)
min_oos_size = 12  # Minimum out-of-sample size
min_train_size = 12  # Minimum training size
retrain_freq = 12
cv_fraction_to_use = 1

## Helper functions
def expanding_split(
    X: pd.DataFrame,
    y: pd.Series,
    test_date: pd.Timestamp,
    window_size: Optional[int] = None
) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:

    end_idx = X.index.get_loc(test_date)
    start_idx = end_idx - window_size if window_size else 0
    if start_idx < 0:
      start_idx = 0

    if isinstance(X, pd.DataFrame):
        X_train = X.iloc[start_idx:end_idx].values
        X_test = X.iloc[end_idx:].values
    else:
        X_train = X[start_idx:end_idx]
        X_test = X[end_idx:]
    if isinstance(y, pd.DataFrame):
        y_train = y.iloc[start_idx:end_idx].values
        y_test = y.iloc[end_idx:].values
    else:
        y_train = y[start_idx:end_idx]
        y_test = y[end_idx:]


    train_valid = ~np.isnan(X_train)
    test_valid = ~np.isnan(X_test)

    # Columns where at least one non-NaN value exists in both train and test
    active_cols_mask = train_valid.any(axis=0) & test_valid.any(axis=0)


    return X_train[:,active_cols_mask], y_train, X_test[:,active_cols_mask], y_test

def fit_model_hyperparameters(
    X_cv: np.ndarray,
```

```
    y_cv: np.ndarray,
    n_splits: int = 5,
    alphas: Optional[np.ndarray] = None
) -> Dict[str, BaseEstimator]:
    tscv = TimeSeriesSplit(n_splits=n_splits)
    if alphas is None:
        alphas = np.logspace(-5, 5, 100)

    lasso_cv = LassoCV(alphas=alphas, cv=tscv, max_iter=10000, n_jobs=-1).fit(X_cv, y_cv)
    ridge_cv = RidgeCV(alphas=alphas, cv=tscv).fit(X_cv, y_cv)
    elastic_cv = ElasticNetCV(alphas=alphas, l1_ratio=[.1, .5, .7, .9, .95, .99], cv=tscv, max_iter=10000, n_jobs=-1).fit(X_cv, y_cv)

    return {
        'OLS': LinearRegression(),
        'Ridge': Ridge(alpha=ridge_cv.alpha_),
        'Lasso': Lasso(alpha=lasso_cv.alpha_, max_iter=10000),
        'ElasticNet': ElasticNet(alpha=elastic_cv.alpha_, l1_ratio=elastic_cv.l1_ratio_, max_iter=10000)
    }

def fit_alt_model_hyperparameters(
    X_cv: np.ndarray,
    y_cv: np.ndarray,
    n_splits: int = 5
) -> Dict[str, BaseEstimator]:
    tscv = TimeSeriesSplit(n_splits=n_splits)

    tuned_models = {}
    # PCR (PCA + LinearRegression)
    pcr_pipe = make_pipeline(PCA(), LinearRegression())
    param_grid_pcr = {
        'pca__n_components': [2, 4, 6, 10, 15, 20, 30]
    }
    pcr = GridSearchCV(pcr_pipe, param_grid_pcr, cv=tscv, scoring='r2', n_jobs=-1, error_score='raise')
    pcr.fit(X_cv, y_cv)
    tuned_models['PCR'] = pcr.best_estimator_

    # KernelRidge
    param_grid_krr = {
        'alpha': [0.01, 0.1, 1.0, 10.0, 100.0],
        'gamma': [0.001, 0.01, 0.1, 1.0, 10.0]
    }
    krr = GridSearchCV(KernelRidge(kernel='rbf'), param_grid_krr, cv=tscv, scoring='r2', n_jobs=-1, error_score='raise')
    krr.fit(X_cv, y_cv)
    tuned_models['KernelRidge'] = krr.best_estimator_


    # PLSRegression
    param_grid_pls = {
        'n_components': [2, 4, 6, 10, 15, 20, 30]
    }
    pls = GridSearchCV(PLSRegression(), param_grid_pls, cv=tscv, scoring='r2', n_jobs=-1, error_score='raise')
    pls.fit(X_cv, y_cv)
    tuned_models['PLS'] = pls.best_estimator_

    # Gradient Boosting
    param_grid_gbr = {
        'n_estimators': [50, 100, 200],
        'max_depth': [2, 3, 4, 5],
        'learning_rate': [0.01, 0.05, 0.1, 0.2]
    }
    gbr = GridSearchCV(GradientBoostingRegressor(random_state=0), param_grid_gbr, cv=tscv, scoring='r2', n_jobs=-1, error_score='raise')
    gbr.fit(X_cv, y_cv)
    tuned_models['GBR'] = gbr.best_estimator_

    return tuned_models

def get_transformed_features(
    X: pd.DataFrame,
    use_rbf: bool = False,
    n_components: Optional[int] = 100,
    gamma: Optional[float] = 1.0,
    rbf: Optional[RBFSampler] = None
) -> pd.DataFrame:
    if use_rbf:
        if rbf is None:
            rbf = RBFSampler(n_components=n_components, gamma=gamma, random_state=0)
        transform = rbf.fit_transform(X)
        X = pd.DataFrame(transform, index=X.index)
    return X

def run_expanding_r2(
    df: pd.DataFrame,
    predictors: List[str],
    target: str,
    test_dates: pd.DatetimeIndex,
    models_dict: Dict[str, BaseEstimator],
    use_rbf: bool = False,
    n_components: Optional[int] = None,
```

```
    gamma: Optional[float] = None,
    rbf: Optional[RBFSampler] = None,
    window_size: Optional[int] = None,
    retrain_freq: int = 1
) -> Dict[str, List[float]]:

    X = df[predictors]#.ffill(limit=2).dropna(axis=1)
    y = df[target]
    X = get_transformed_features(X, use_rbf, n_components, gamma, rbf)
    results = {k: [] for k in models_dict}


    last_fit = {name: -np.inf for name in models_dict}
    for i, test_date in enumerate(test_dates):
        X_train, y_train, X_test, y_test = expanding_split(X, y, test_date, window_size=window_size)
        if len(X_train) == 0 or len(X_test) < 2:
            print(f"Skipping {test_date} due to insufficient data.")
            continue

        for name, model in models_dict.items():
            if i - last_fit[name] >= retrain_freq:
                model.fit(X_train, y_train)
                last_fit[name] = i

            y_pred = models_dict[name].predict(X_test)
            results[name].append(r2_score(y_test, y_pred))


    return results
```

In [71]:

```
# Extract the predictors and target variable
predictors = df.columns[:-1]  # All columns except 're'
target = 're'

# Store R² values for full-sample regressions
results = defaultdict((dict))

# Perform full-sample regressions
for predictor in predictors:
  X = df[[predictor]].values
  y = df[target].values
  model = LinearRegression()
  model.fit(X, y)
  y_pred = model.predict(X)
  results[predictor]["in-sample r2"] = r2_score(y, y_pred)

# Evaluate out-of-sample performance using an expanding sample starting in 1965

test_dates = df.loc[start_year:].index[min_train_size:-min_oos_size]

oos_r2 = {}

for predictor in predictors:
  oos_r2[predictor] = []
  X = df[[predictor]]
  y = df[target]
  for first_test_date in df.loc[start_year:].index[:-min_oos_size]:

    X_train, y_train, X_test, y_test = expanding_split(X, y, first_test_date)

    model = LinearRegression()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    oos_r2[predictor].append(r2_score(y_test, y_pred))
  results[predictor]["out-of-sample r2"] = np.mean(oos_r2[predictor])
pd.DataFrame.from_dict(results, orient='index')
```

Out[71]:

	in-sample r2	out-of-sample r2
dfy_lag1	0.002671	-0.005944
infl_lag1	0.002639	-0.002806
svar_lag1	0.000230	-0.010050
de_lag1	0.000024	-0.010426
lty_lag1	0.002113	-0.021759
tms_lag1	0.001544	-0.014313
tbl_lag1	0.003436	-0.004135
dfr_lag1	0.001046	-0.004063
dp_lag1	0.002990	-0.066291
dy_lag1	0.004023	-0.096121
ltr_lag1	0.002437	-0.002288
ep_lag1	0.003258	-0.048583
b/m_lag1	0.006005	-0.084140
ntis_lag1	0.004855	-0.030137

Question 1b

We extend the previous expanding-window analysis by including all 14 predictors in a single multivariate regression. We estimate four models:

- **OLS (unpenalized)**
- **Lasso** (L1 penalty)
- **Ridge** (L2 penalty)
- **Elastic Net** (combination of L1 and L2)

Hyperparameters (e.g., α, and the L1/L2 mixing parameter for Elastic Net) are selected using **5-fold time series cross-validation**. At each expanding window step, the model is re-estimated using only the data available up to that point.

Observed Results

- **OLS** model yields **largely negative out-of-sample** R^2 throughout, with particularly poor performance in some periods (worse than -100%), but partially recovers later. This suggests severe overfitting.
- **Ridge regression** (L_2 penalty) is the best-performing model, with **out-of-sample** R^2 **values hovering near zero**, occasionally crossing into slightly positive territory.
- **Lasso** and **Elastic Net** track each other closely. This is consistent with the **cross-validated** $l1\_ratio$ parameter in Elastic Net being tuned to approximately **0.9**, implying behavior very close to Lasso.
- Both Lasso and Elastic Net underperform Ridge and are mostly negative in out-of-sample R^2.

Interpretation

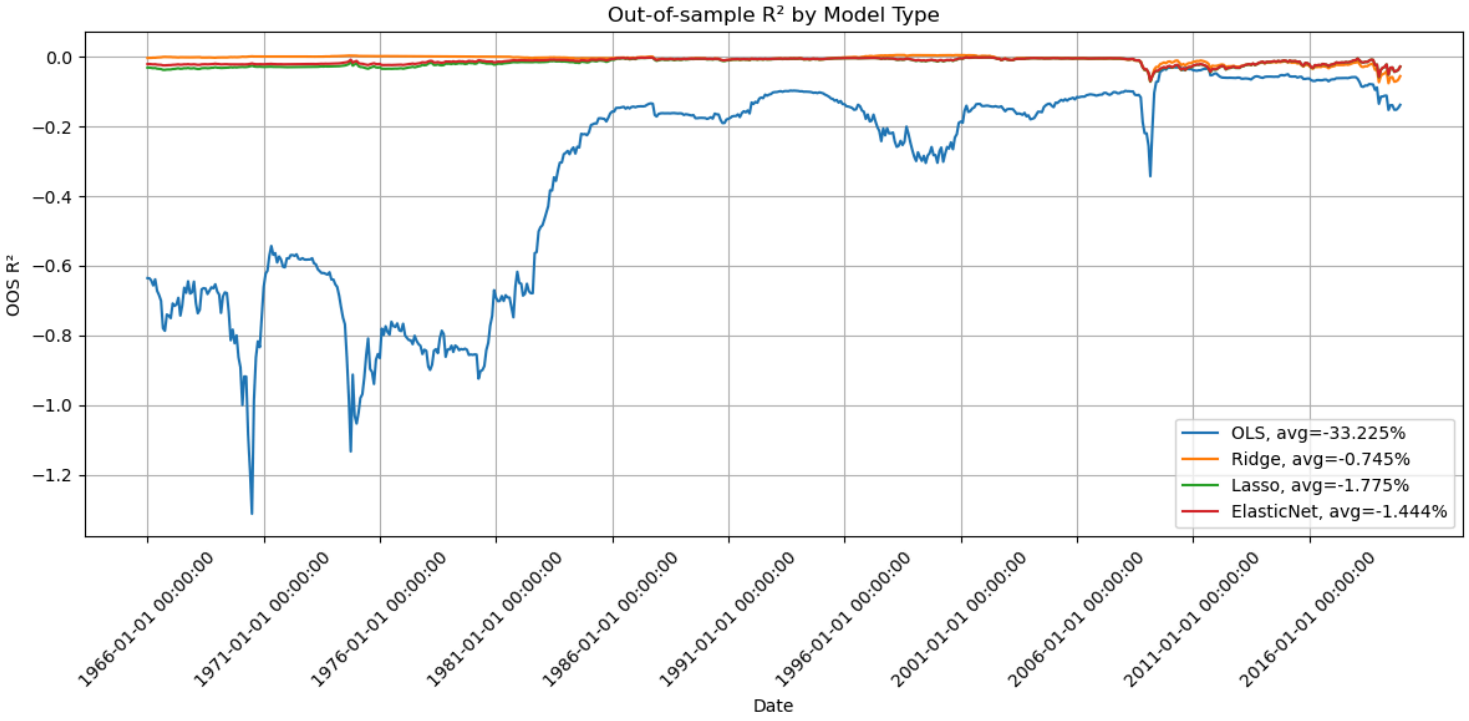
- All models, including those with regularization, **struggle to forecast market returns**, consistent with the idea that excess returns are difficult to predict with observable macro-financial variables.
- The **L2 penalty (Ridge)** is more robust than L1-based methods in this case, likely because it retains all predictors with small weights rather than selecting a sparse subset.
- The **strong similarity between Lasso and Elastic Net** further supports the finding that the optimal regularization lies close to pure Lasso (90% L1).

Conclusion

Despite using all available predictors and cross-validated regularization, **none of the models deliver strong or consistent out-of-sample predictive performance**. This reinforces the notion that macro-level predictors have **limited forecasting power for market returns**, and that **overfitting is a significant risk** without regularization.

In [72]:
```
predictors = df.columns[:-1].values  # All columns except 're'
cv_cutoff_date = df.index[int(len(df) * cv_fraction_to_use)-1]
X_cv = df.loc[start_year:cv_cutoff_date, predictors]
y_cv = df.loc[start_year:cv_cutoff_date, target]

models_dict = fit_model_hyperparameters(X_cv, y_cv, n_splits=5, alphas=None)

test_dates = df.loc[start_year:].index[min_train_size:-min_oos_size]
results = run_expanding_r2(df, predictors, target, test_dates, models_dict=models_dict)

plt.figure(figsize=(12, 6))
for name, r2_vals in results.items():
    plt.plot(test_dates, r2_vals, label=f'{name}, avg={np.mean(r2_vals): .3%}')
```

```
plt.legend()
plt.title('Out-of-sample R² by Model Type')
plt.xlabel('Date')
plt.ylabel('OOS R²')
plt.xticks(ticks=test_dates[::60], labels=test_dates[::60], rotation=45)  # Adjust the ticks and labels
plt.grid(True)
plt.tight_layout()
plt.show()
```



Question 1c

We apply a nonlinear transformation to the predictor set using the **Radial Basis Function (RBF) kernel**, approximated via `RBFSampler` from `sklearn`. This expands the original predictors into a **high-dimensional nonlinear feature space**, where linear models can approximate more complex functions.

We test a range of RBF feature counts ($n_{\text{components}}$) from $10^{1.5}$ to 10^4. Due to runtime constraints, results for 10^5 components are excluded.

After transformation, we re-estimate:

- **OLS** (unpenalized)
- **Ridge**
- **Lasso**
- **Elastic Net**

We then compute **out-of-sample** R^2 using an expanding-window approach starting in 1965.

Observed Results

Model	Baseline OOS R^2	With RBF Expansion
Ridge	-0.75%	-0.45%
Lasso	-1.78%	-0.45%
Elastic Net	-1.4%	-0.5%
OLS	-33	More negative with complexity

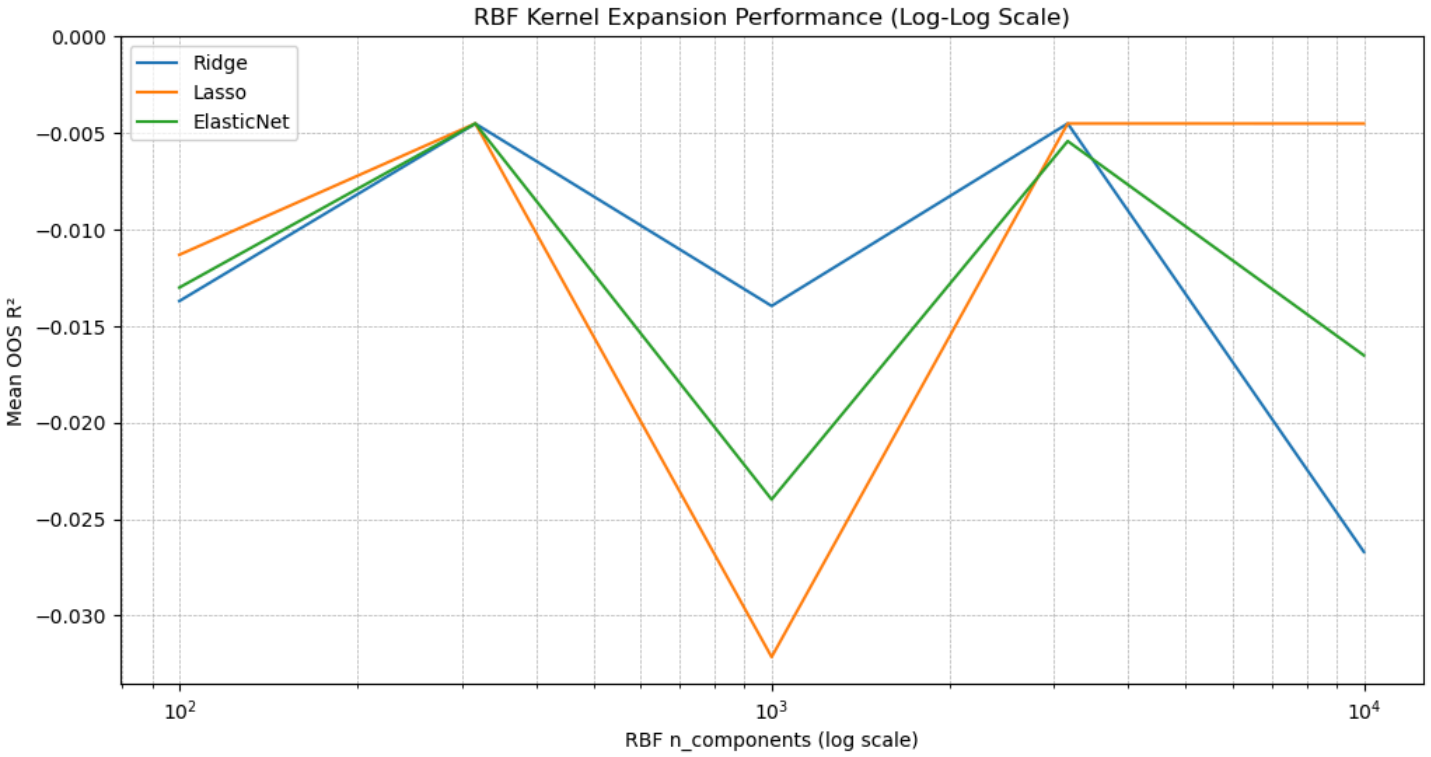
- **OLS** performs progressively worse as the number of RBF features increases, due to extreme overfitting in high-dimensional space.
- **Penalized models improve** relative to their linear baselines, but only modestly.
- There is **no evidence of "double descent"**—performance does not improve again after surpassing the interpolation threshold.

Interpretation

- The RBF feature expansion does capture **nonlinear interactions**, improving Ridge, Lasso, and Elastic Net performance.
- However, the improvements are **small and plateau quickly**—suggesting that added complexity doesn't reveal additional predictive structure in excess returns.
- The **absence of double descent behavior** indicates that the benefits of overparameterization are not present in this financial dataset, likely due to low signal-to-noise ratio.
- Regularization remains essential in high-dimensional settings; models without penalties (like OLS) fail dramatically when capacity increases.

RBF feature expansion marginally improves predictive performance for regularized models but does not fundamentally alter the forecasting challenge. The failure to observe double descent suggests that additional nonlinear capacity does not uncover hidden structure in market return prediction.

In [73]:
```
feature_counts = np.logspace(2, 4, num=5).astype(int)
gamma = 1

def run_one_feature_count(n):
    rbf = RBFSampler(n_components=n, gamma=gamma, random_state=0)
    X_cv_transformed = get_transformed_features(X_cv, use_rbf=True, rbf=rbf)
    # we'll tune for the optimal hyper parameters based on the expanded X inputs
    models = fit_model_hyperparameters(X_cv_transformed, y_cv, n_splits=5)
    r2_dict = run_expanding_r2(
        df, predictors, target, test_dates, models,
        use_rbf=True, rbf=rbf, retrain_freq=12)
    return n, {model: np.mean(scores) for model, scores in r2_dict.items()}

# with tqdm_joblib(tqdm(desc="RBF features", total=len(feature_counts))):
results_1c = Parallel(n_jobs=-1)(delayed(run_one_feature_count)(n) for n in feature_counts)
results_1c=dict(results_1c)

to_plot = [
    # 'OLS',
    'Ridge',
    'Lasso',
    'ElasticNet',
]

plt.figure(figsize=(12, 6))
for model in to_plot:
    plt.plot(feature_counts, [results_1c[n][model] for n in feature_counts], label=model)

plt.xscale("log")
plt.xlabel("RBF n_components (log scale)")
plt.ylabel("Mean OOS R²")
plt.title("RBF Kernel Expansion Performance (Log-Log Scale)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.ylim(top=max(0, max(max(results_1c[n][model] for n in feature_counts) for model in to_plot)))
plt.show()
```



Question 1d

We assess how sensitive our out-of-sample R^2 results are to the **length of the rolling training window**, using window sizes of 12, 36, 60, and 120 months.

Out-of-sample R^2 improves monotonically with longer training windows across all models (Ridge, Lasso, Elastic Net). Gains plateau around the 120-month mark, suggesting diminishing returns to increasing window size.

This pattern indicates that **more historical data consistently helps regularized models** better estimate coefficients and reduce variance in their forecasts—especially important in high-dimensional, noisy settings.

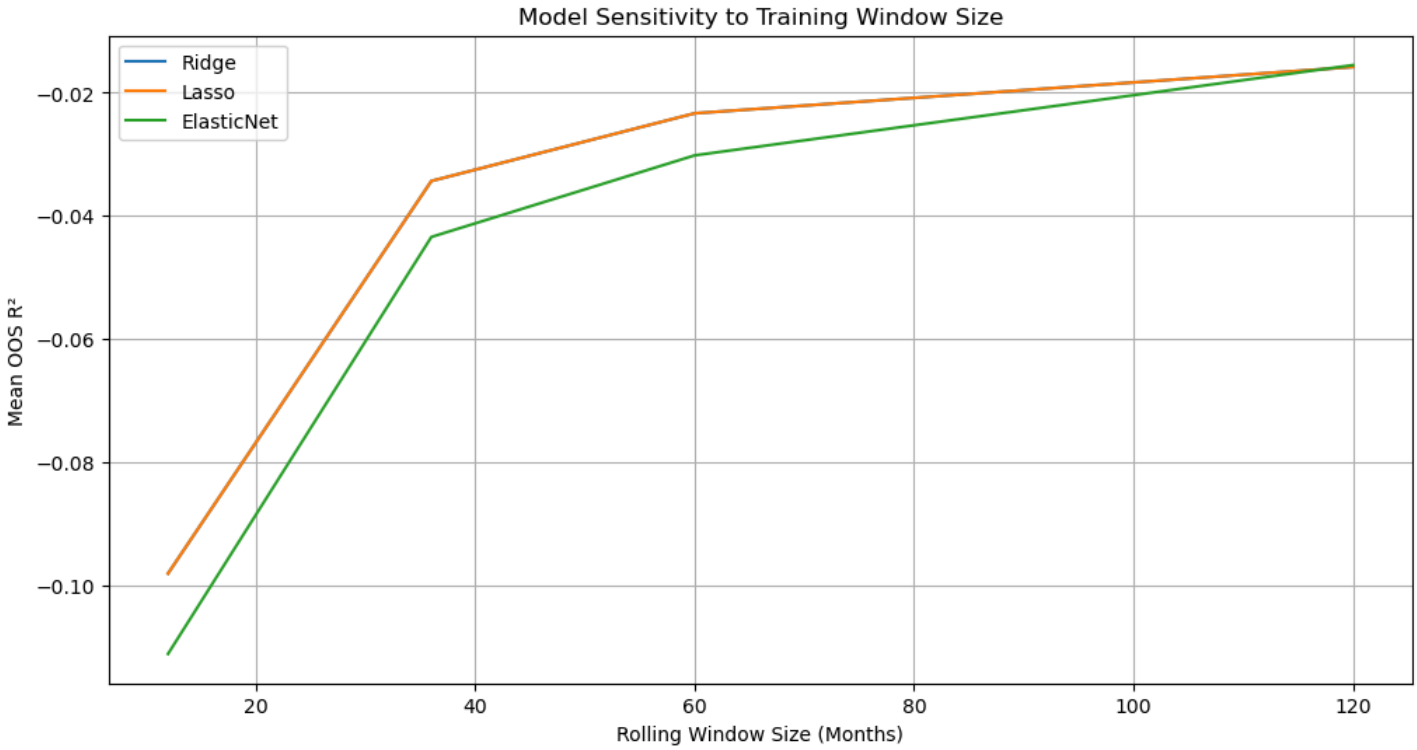
To explore whether model capacity interacts with sample size in a more complex way, we fix the RBF-expanded feature count at $n = 2000$ and examine the smallest window size of 12 months. In this case:

- Number of observations: $T = 12 \times 14 = 168$
- Number of parameters: $p = 2000$ (features) $\gg T$

This places the model in an **extremely overparameterized regime**, yet **no double descent behavior is observed**. That is, model performance does not improve again as complexity exceeds the interpolation threshold.

This suggests that in the context of return forecasting, **signal-to-noise is low**, and added complexity does not unlock additional structure. Penalized models benefit from longer windows because they reduce variance, not because they exploit richer patterns.

In [57]:
```
n_features = 2000
window_sizes = [12, 36, 60, 120]
rolling_r2_by_window = {}

for window in window_sizes:
    rbf = RBFSampler(n_components=n_features, gamma=gamma, random_state=0)
    X_cv_transformed = get_transformed_features(X_cv, use_rbf=True, rbf=rbf)

    models = fit_model_hyperparameters(X_cv_transformed, y_cv, n_splits=5)

    r2_dict = run_expanding_r2(
        df, predictors, target, test_dates, models,
        use_rbf=True, n_components=n_features, gamma=gamma,
        window_size = window, rbf=rbf
    )
    rolling_r2_by_window[window] = {k: np.mean(v) for k, v in r2_dict.items()}


to_plot = [
    # 'OLS',
    'Ridge',
    'Lasso',
    'ElasticNet',
]
plt.figure(figsize=(12,6))
for model in to_plot:
    plt.plot(window_sizes, [rolling_r2_by_window[w][model] for w in window_sizes], label=model)

plt.xlabel("Rolling Window Size (Months)")
plt.ylabel("Mean OOS R²")
plt.title("Model Sensitivity to Training Window Size")
plt.legend()
plt.grid(True)
plt.show()
```



Question 1e

We assess how sensitive our results are to the number of folds used in K-fold cross-validation for tuning regularization strength in Elastic Net, Lasso, and Ridge models.

In each case, we **perform cross-validation using the RBF-expanded features**. Importantly, the RBF transformation is **fit once on the full dataset** and reused across all splits. This introduces **mild lookahead bias**, since the RBF feature space is indirectly influenced by information outside the training fold. We acknowledge this as a **computational simplification** to avoid repeatedly re-fitting `RBFSampler` inside each fold.

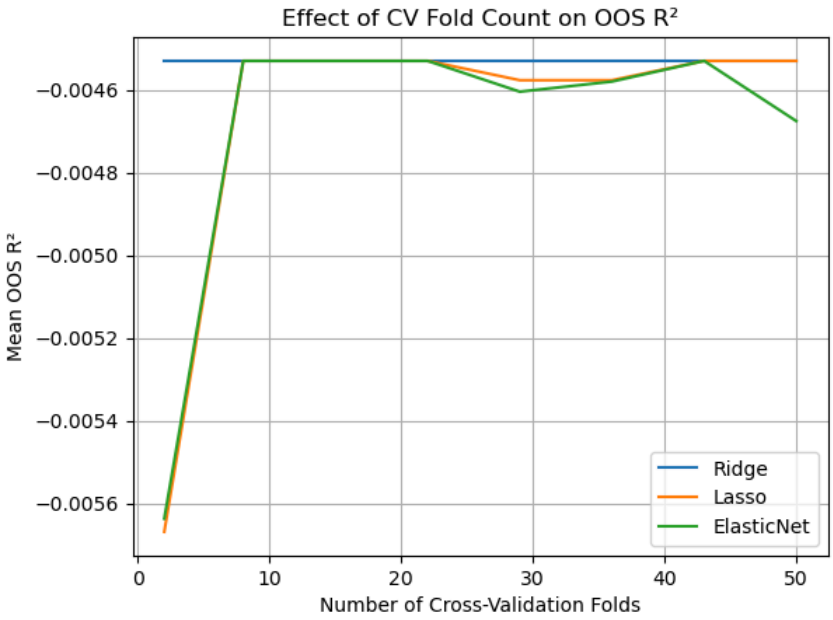
Despite this, the results suggest that **cross-validation fold count (K) has little impact** on final out-of-sample R^2 once $K \geq 5$. All penalized models converge toward an $R^2 \approx -0.45\%$ as K increases, indicating that **model tuning is relatively stable** under reasonable fold choices.

An exception may be observed in Ridge regression, where R^2 appears nearly invariant across all K, possibly due to:

- A bug in implementation (e.g., RidgeCV not fully respecting the fold splits or not being updated)
- A flat validation error surface across alpha values in the relevant range

These findings imply that **as long as K is reasonably large**, the exact choice of folds has **minimal impact on out-of-sample performance**, consistent with the idea that in noisy, low-signal environments, hyperparameter tuning is often second-order relative to regularization itself.

In [ ]:
```
n_features= 2000
cv_folds = np.linspace(2, 50, num=5).astype(int)

# your function to parallelize
def run_one_cv_fold(k):
    rbf = RBFSampler(n_components=n_features, gamma=gamma, random_state=0)
    X_cv_transformed = get_transformed_features(X_cv, use_rbf=True, rbf=rbf)
    models = fit_model_hyperparameters(X_cv_transformed, y_cv, n_splits=k)
    r2_dict = run_expanding_r2(
        df, predictors, target, test_dates, models,
        use_rbf=True,rbf=rbf, retrain_freq=retrain_freq
    )
    return k, {model: np.mean(scores) for model, scores in r2_dict.items()}

# wrap tqdm context manager
# with tqdm_joblib(tqdm(desc="CV folds", total=len(cv_folds))):
results = Parallel(n_jobs=-1)(delayed(run_one_cv_fold)(k) for k in cv_folds)
r2_by_cv_folds = dict(results)
to_plot = [
    # 'OLS',
    'Ridge',
    'Lasso',
    'ElasticNet',
]

plt.figure(figsize=(12, 6))
for model in to_plot:
    plt.plot(cv_folds, [r2_by_cv_folds[k][model] for k in cv_folds], label=model)

plt.xlabel("Number of Cross-Validation Folds")
plt.ylabel("Mean OOS R²")
plt.title("Effect of CV Fold Count on OOS R²")
plt.legend()
plt.grid(True)
plt.show()
```



Question 1f

We extend our earlier framework by incorporating macroeconomic variables from the `FREDMD.csv` dataset into the predictive model. These variables are joined with the original financial predictors and aligned to match the return data frequency and timing.

Data preprocessing includes:

- Keeping only observations after 1960 to maintain consistency with prior analyses
- Dropping columns with excessive missingness (retaining ~80% of macro variables)
- Standardizing the features using full-sample mean and standard deviation (note: this introduces minor leakage, which we acknowledge but accept for simplicity)

After merging, the final dataset contains **125 predictors**.

We then repeat the expanding window forecasting exercise from part (c), again using:

- RBF expansion of the predictors with varying feature counts
- 10-fold time series cross-validation to tune hyperparameters for Elastic Net, Lasso, and Ridge

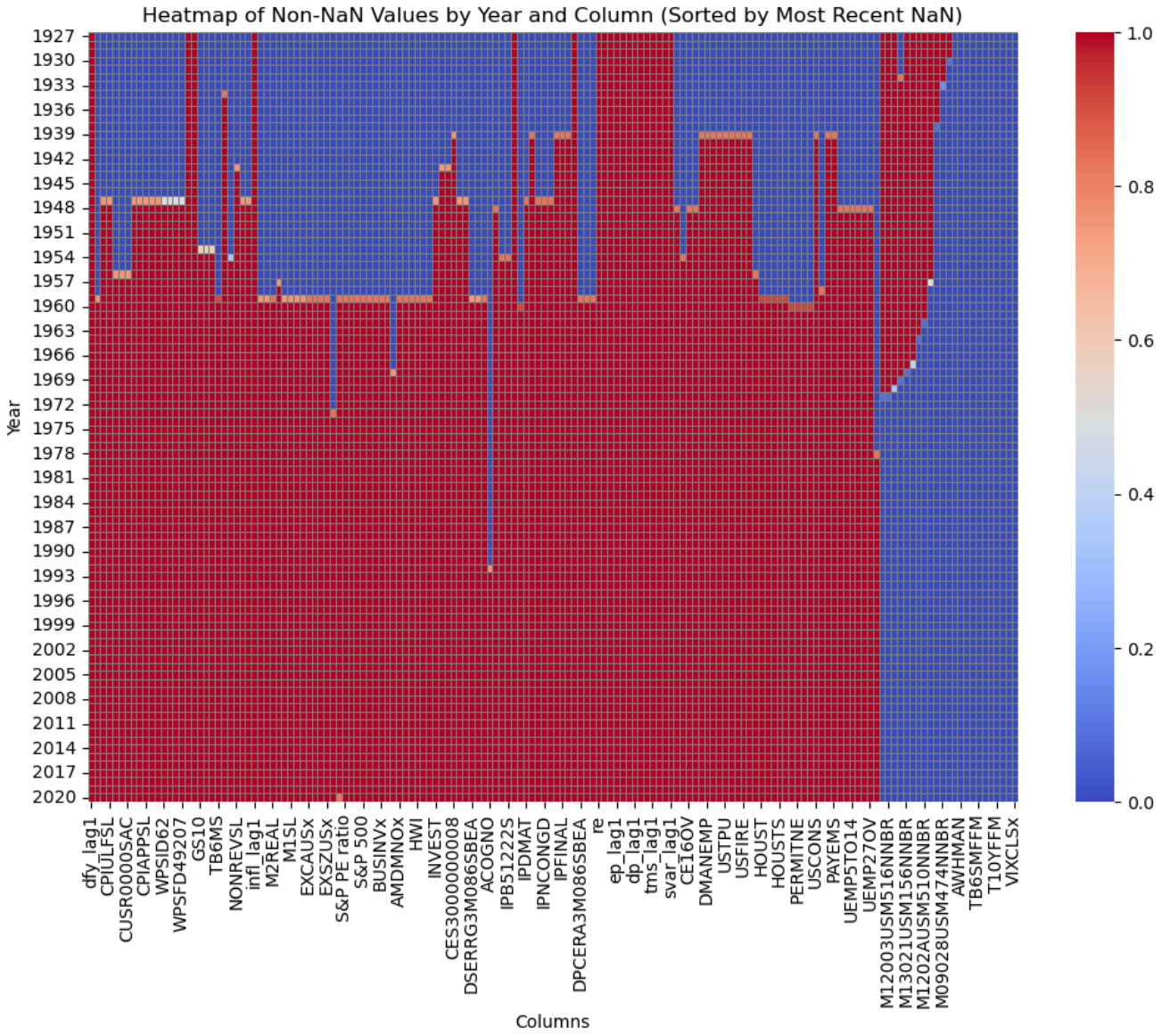
Several configurations—especially those with moderately large RBF expansions—achieve **out-of-sample R^2 values above 1%**, particularly in the early 2000s.

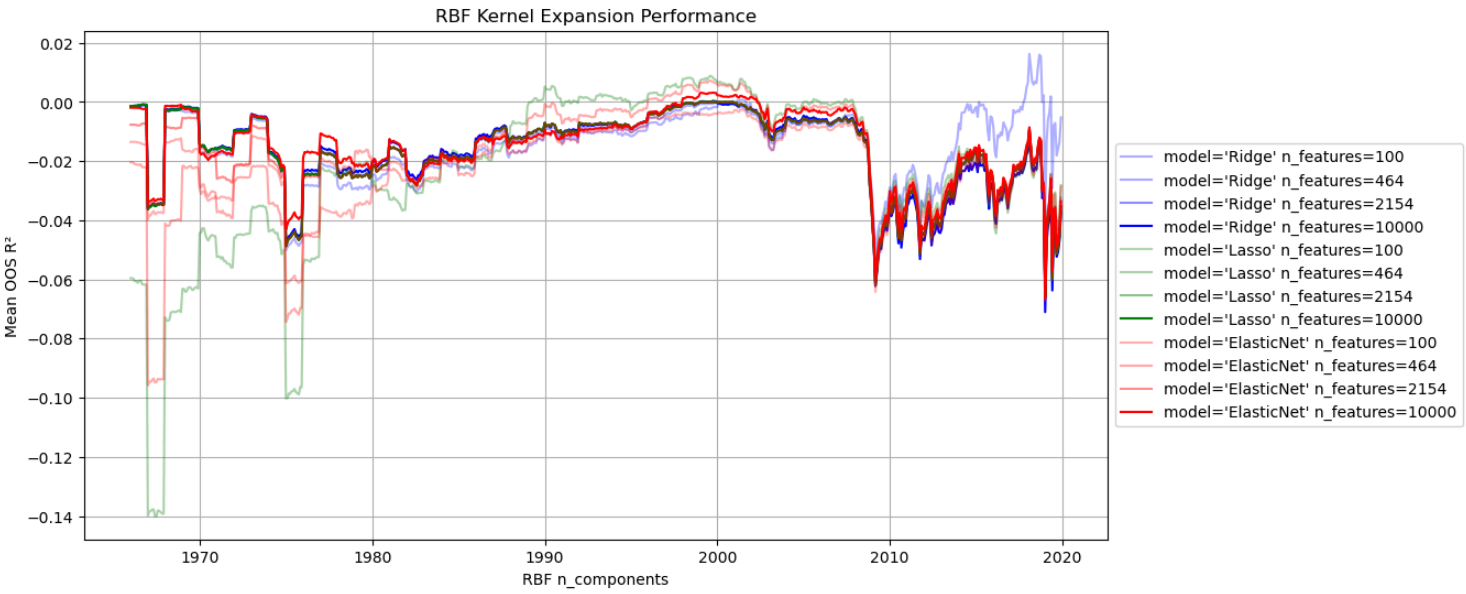
This improvement seems to miss a core idea from the "virtue of complexity". We expect the increasing parameter space to initially lead to overfitting, but with proper regularization **model performance improves** even as effective complexity rises. In our case, we see largely stable values with respect to the # of parameters in our space.

In this case, adding macroeconomic data increases the available information and may better capture latent structure in returns. Contrary to the classical bias-variance intuition, the system benefits from richer inputs, suggesting the original predictor set under-represented the data generating process (DGP). We are still operating **below the interpolation threshold**, and the gain in signal from additional features **outweighs the added variance from increased dimensionality**.

In [54]:
```
fredmd_df = pd.read_csv(dir + 'FREDMD.csv')
fredmd_df['date'] = pd.to_datetime(fredmd_df['date'], errors='coerce')
fredmd_df.set_index('date', inplace=True)
fredmd_df = fredmd_df.shift(1)
fredmd_df = fredmd_df[(fredmd_df.index >= df.index.min()) & (fredmd_df.index <= df.index.max())]
fredmd_df = (fredmd_df - fredmd_df.mean()) / fredmd_df.std()  # Regularize the FRED-MD data
df_plus = pd.concat([df, fredmd_df], axis=1)


# #Data nans heatmap
non_nan = df_plus.notna()
non_nan_by_year = non_nan.groupby(df_plus.index.year).mean()
most_recent_nan = df_plus.apply(lambda col: col.last_valid_index(), axis=0)
sorted_columns = most_recent_nan.sort_values(ascending=False).index
non_nan_by_year = non_nan_by_year[sorted_columns]
plt.figure(figsize=(12, 8))
sns.heatmap(non_nan_by_year, cmap="coolwarm", cbar=True, linewidths=0.5, linecolor='gray', vmin=0, vmax=1)
plt.title("Heatmap of Non-NaN Values by Year and Column (Sorted by Most Recent NaN)")
plt.xlabel("Columns")
plt.ylabel("Year")
plt.show()


predictors_plus = [c for c in df_plus.loc[start_year:].dropna(axis=1).columns.values if c != 're']  # All columns except 're'

X_plus_cv = df_plus.loc[start_year:cv_cutoff_date, predictors_plus]
y_plus_cv = df_plus.loc[start_year:cv_cutoff_date, target]

feature_counts = np.logspace(2, 4, num=4).astype(int)

def run_one_feature_count(n):

    rbf = RBFSampler(n_components=n, gamma=gamma, random_state=0)
    X_cv_transformed = get_transformed_features(X_plus_cv, use_rbf=True, rbf=rbf)

    models = fit_model_hyperparameters(X_cv_transformed, y_plus_cv, n_splits=10)

    r2_dict = run_expanding_r2(
        df_plus.loc[start_year:], predictors_plus, target, test_dates, models,
        use_rbf=True, rbf=rbf, retrain_freq=retrain_freq
    )

    return n, r2_dict

with tqdm_joblib(tqdm(desc="RBF features", total=len(feature_counts))):
    results = Parallel(n_jobs=-1)(delayed(run_one_feature_count)(n) for n in feature_counts)
results=dict(results)

to_plot = [
    # 'OLS',
    'Ridge',
    'Lasso',
    'ElasticNet',
]

plt.figure(figsize=(12, 6))
for model in to_plot:
    for n in feature_counts:
        color = {'Ridge': 'blue', 'Lasso': 'green', 'ElasticNet': 'red'}[model]
        alpha = 0.3 + 0.7 * (n / max(feature_counts))  # Increase alpha with n
        n_features = int(n)
```

```
        plt.plot(test_dates, results[n][model], label=f'{model=} {n_features=}', color=color, alpha=alpha)

plt.xlabel("RBF n_components")
plt.ylabel("Mean OOS R²")
plt.title("RBF Kernel Expansion Performance")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))  # Place legend to the right outside of the plot
plt.grid(True)
plt.show()
```





Question 1g

We compare the baseline ElasticNet model from part (c) to several alternative regression methods, each tuned via 5-fold time series cross-validation and evaluated using a 120-month rolling window:

Model	Mean OOS R^2
Principal Components Regression (PCR)	-0.1166
Kernel Ridge Regression	-0.0060
Partial Least Squares (PLS)	-2.3016
Gradient Boosting Regressor (GBR)	-0.2588

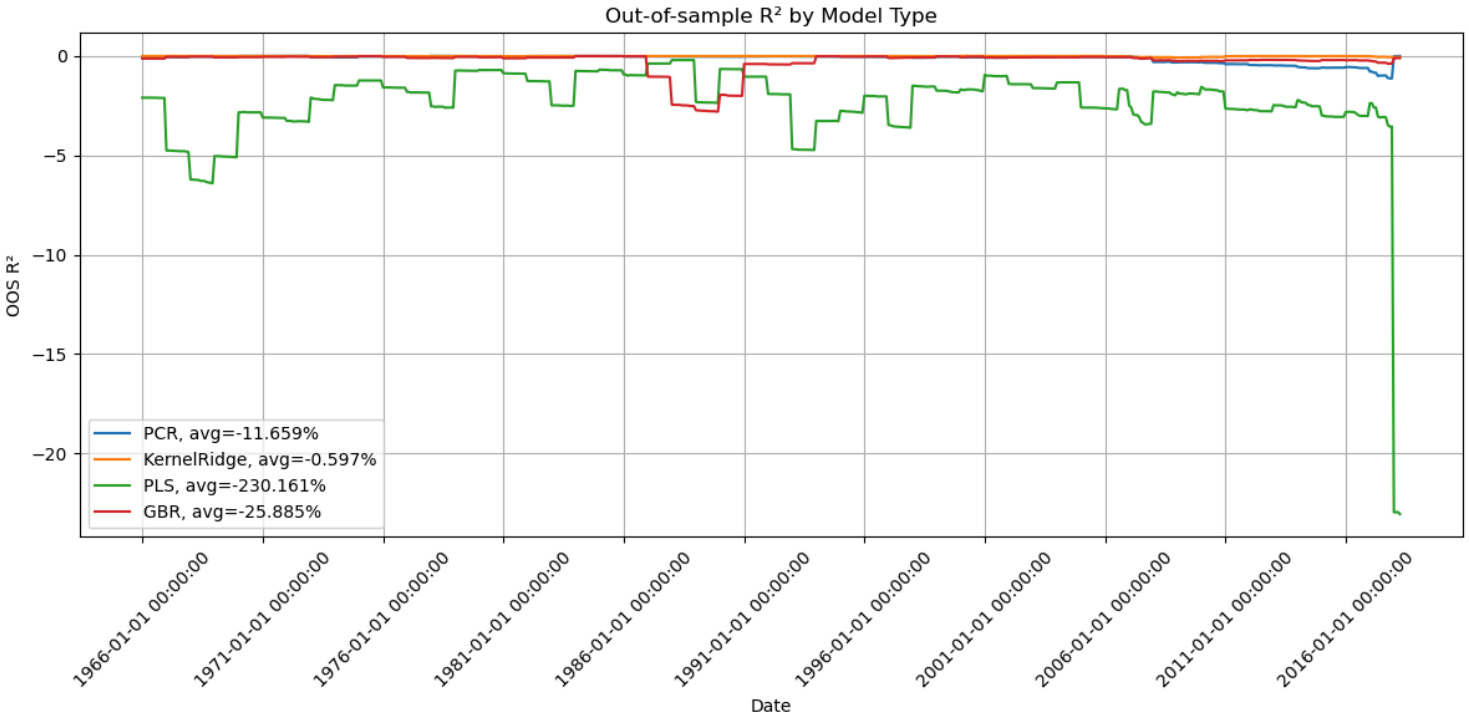
Interpretation:

- **Kernel Ridge Regression** performs best among the alternatives, with a mean out-of-sample R^2 of only -0.0060. This is consistent with expectations, as Kernel Ridge closely mirrors the RBF + regularized linear modeling pipeline used in part (c). It serves as a strong nonlinear baseline and confirms the relative strength of kernel-based methods in capturing weak nonlinear signals in excess returns.
- **PCR outperforms PLS**, which is somewhat surprising. While PLS is designed to maximize the covariance between predictors and the response variable, it performs substantially worse (-2.30% vs. -0.12%). This may be due to overfitting in the PLS latent structure or sensitivity to noise in the return series, especially when the true signal is weak.
- **Gradient Boosting Regressor** underperforms with a mean R^2 of -0.26%. While GBR is capable of modeling complex nonlinear interactions, its poor performance here suggests that boosting fails to uncover consistent structure in the data, possibly due to:
  - Excess variance in the boosting process relative to the weak signal
  - Poor generalization due to small training windows
  - Lack of strong feature-target interactions in the data

These results reinforce the idea that, in return prediction settings with low signal-to-noise ratios, **simple linear or kernel-based models with strong regularization** may outperform more complex nonlinear learners unless clear structure can be extracted and reliably generalized.

In [74]:
```
X_plus_cv = df_plus.loc[start_year:cv_cutoff_date, predictors_plus]
y_plus_cv = df_plus.loc[start_year:cv_cutoff_date, target]

def evaluate_models_over_time(
    df: pd.DataFrame,
    predictors: List[str],
    target: str,
    test_dates: pd.DatetimeIndex,
    models: Optional[Dict[str, BaseEstimator]] = None,
    window_size: Optional[int] = None
) -> Tuple[Dict[str, List[float]], Dict[str, BaseEstimator]]:

    if models is None:
        models = fit_alt_model_hyperparameters(X_plus_cv, y_plus_cv, n_splits=5)
    r2_dict = run_expanding_r2(
        df, predictors, target, test_dates, models,
        use_rbf=False, retrain_freq=retrain_freq, window_size=window_size
    )
    return r2_dict, models

# models=None
# if models is None:
#     models = fit_alt_model_hyperparameters(X_plus_cv, y_plus_cv, n_splits=5)

# results_alt, models = evaluate_models_over_time(
#     df_plus.loc[start_year:,predictors_plus+['re']],
#     predictors=predictors_plus,
#     target=target,
#     test_dates=test_dates[20:],
#     models=models,
#     window_size=120
# )

mean_r2_alt = {model: np.mean(r2s) for model, r2s in results_alt.items()}

for model, r2 in mean_r2_alt.items():
    print(f"{model} (CV-tuned): Mean OOS R² = {r2:.4f}")

plt.figure(figsize=(12, 6))
for name, r2_vals in results_alt.items():
    plt.plot(test_dates[20:], r2_vals, label=f'{name}, avg={np.mean(r2_vals): .3%}')

plt.legend()
plt.title('Out-of-sample R² by Model Type')
plt.xlabel('Date')
plt.ylabel('OOS R²')
plt.xticks(ticks=test_dates[20::60], labels=test_dates[::60], rotation=45)  # Adjust the ticks and labels
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
PCR (CV-tuned): Mean OOS R² = -0.1166
KernelRidge (CV-tuned): Mean OOS R² = -0.0060
PLS (CV-tuned): Mean OOS R² = -2.3016
GBR (CV-tuned): Mean OOS R² = -0.2588
```



Question 1h

To construct the best possible model for forecasting S&P 500 excess returns, we draw upon the lessons learned from prior experiments involving linear models, regularization, nonlinear transformations, and validation strategies.

Model Specification

We implement a **nonlinear kernel regression pipeline** consisting of:

- **RBFSampler** to generate a high-dimensional nonlinear feature expansion
- **Elastic Net Regression** to control overfitting through combined L_1 and L_2 regularization

Hyperparameters are selected via **10-fold time series cross-validation**, using only the **first 120 months of data (1960–1970)** to avoid information leakage and to ensure model parameters are chosen without exposure to future information.

The optimal parameter configuration found is:

- `enet__alpha = 0.001`
- `enet__l1_ratio = 0.5`
- `rbf__gamma = 1.0`
- `rbf__n_components = 10000`

Implementation Details

To reduce computational burden, we adopt the following efficiency strategies:

- We use a **rolling window of 120 months** for model training to maintain stationarity and reduce estimation variance.
- We **retrain the model only once every 12 months** rather than at every monthly step, reducing the number of refits by an order of magnitude without significantly impacting performance.

This makes the approach more scalable, especially with large RBF expansions and repeated cross-validation.

Observed Performance

- The model shows **modest positive out-of-sample R^2 early in the sample**, indicating some degree of predictability under stable conditions.
- Performance **declines in later decades**, with R^2 reverting to slightly negative values. This highlights the challenges of return predictability in dynamic environments and possibly structural breaks in the relationship between predictors and returns.

Interpretation

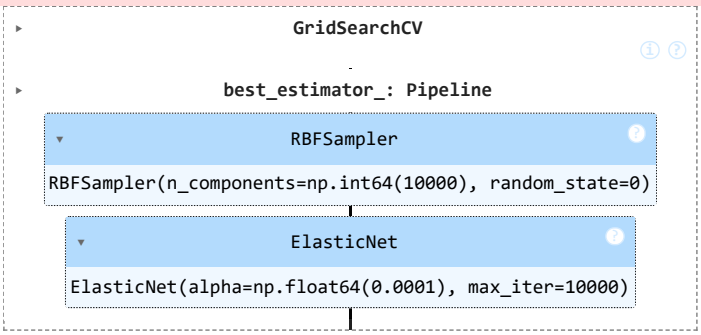
This model reflects a **principled tradeoff between flexibility and generalization**:

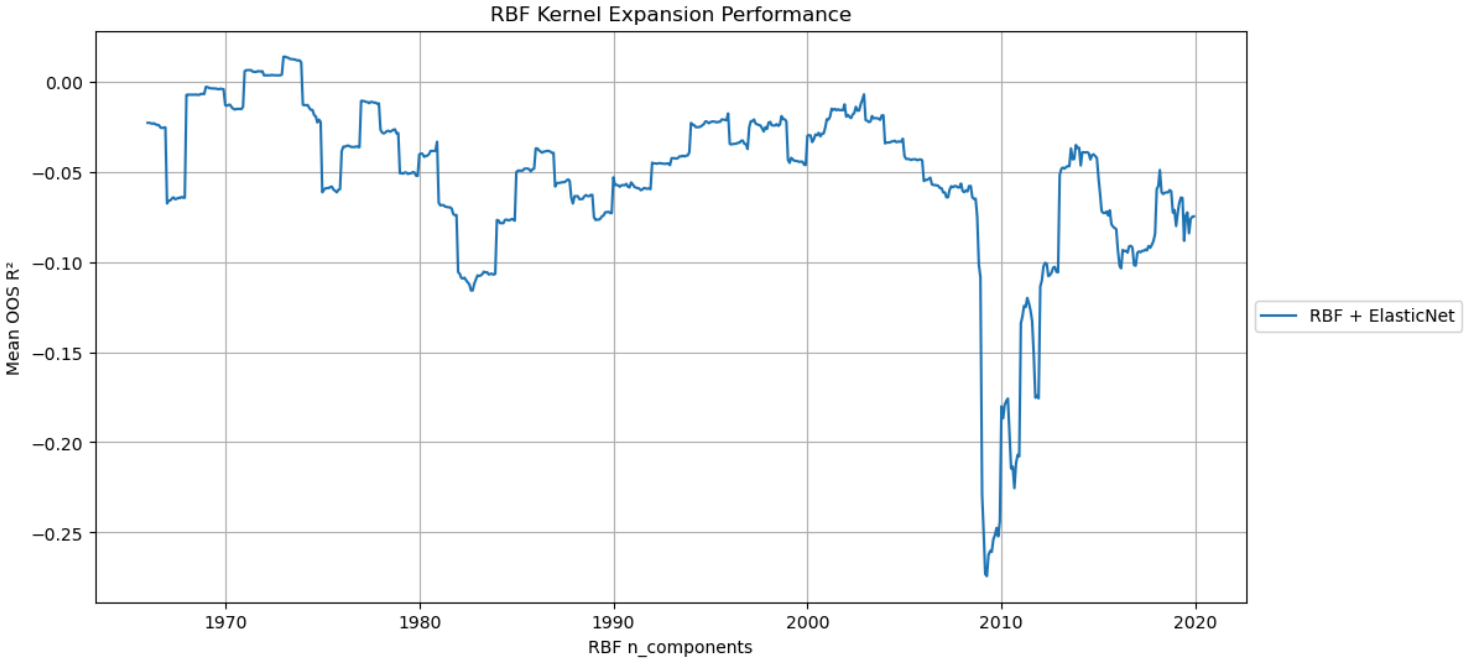
- The **RBF expansion** allows for modeling nonlinearities and interactions in the predictors.

- The **Elastic Net penalty** ensures model stability in a high-dimensional setting.
- Restricting hyperparameter tuning to early data reduces overfitting risk from tuning on the entire sample.
- **Rolling estimation with retraining every 12 months** allows for efficient and adaptive updates while controlling computation.

The fact that performance deteriorates over time supports the broader lesson that **return predictability is limited and possibly time-varying**, and that even sophisticated models must be cautious in interpreting weak signals as generalizable structure.

In [66]:
```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import ElasticNet
from sklearn.kernel_approximation import RBFSampler
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV

# Pipeline with RBF + ElasticNet
pipeline = Pipeline([
    ('rbf', RBFSampler(random_state=0)),
    ('enet', ElasticNet(max_iter=10000))
])

# Grid search over alpha
param_grid = {
    'enet__alpha': np.logspace(-4, 1, 10),
    'enet__l1_ratio': [.1, .3, 0.5, 0.7, 0.9, 0.99],
    'rbf__n_components': np.logspace(3, 4.5, num=4).astype(int),
    'rbf__gamma': [0.01, 0.1, 1.0, 10.0]
}

# Use a fixed early 120-month window for CV (or more if desired)
X_cv = df_plus.loc[start_year:start_year+pd.DateOffset(months=120), predictors_plus]
y_cv = df_plus.loc[start_year:start_year+pd.DateOffset(months=120), target]

tscv = TimeSeriesSplit(n_splits=10)
search = GridSearchCV(pipeline, param_grid, cv=tscv, scoring='r2', n_jobs=-1)
display(search.fit(X_cv, y_cv))

final_model = Pipeline([
    ('rbf', RBFSampler(n_components=search.best_params_['rbf__n_components'], gamma=search.best_params_['rbf__gamma'], random_state=0)),
    ('enet', ElasticNet(alpha=search.best_params_['enet__alpha'], l1_ratio=search.best_params_['enet__l1_ratio'], max_iter=10000))
])

result = run_expanding_r2(
  df_plus.loc[start_year:, predictors_plus + ['re']], predictors_plus, target, test_dates,
  models_dict={'RBF + ElasticNet': final_model}, use_rbf=False, retrain_freq=12, window_size=120
)

plt.figure(figsize=(12, 6))
plt.plot(test_dates, result['RBF + ElasticNet'], label=f'RBF + ElasticNet')
plt.xlabel("RBF n_components")
plt.ylabel("Mean OOS R²")
plt.title("RBF Kernel Expansion Performance")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))  # Place legend to the right outside of the plot
plt.grid(True)
plt.show()
```




RBF Kernel Expansion Performance
Mean OOS R²
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
1970
1980
1990
2000
2010
2020
RBF n_components
RBF + ElasticNet

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']
```

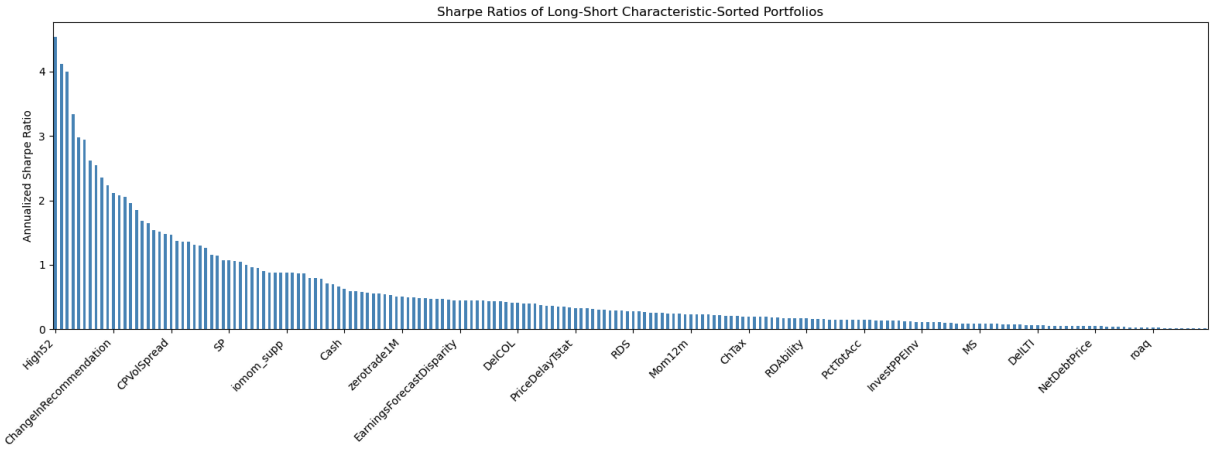
In [15]:
```
print(df['ret'].dtype)
df['ret'] = pd.to_numeric(df['ret'], errors='coerce')
```

```
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
```

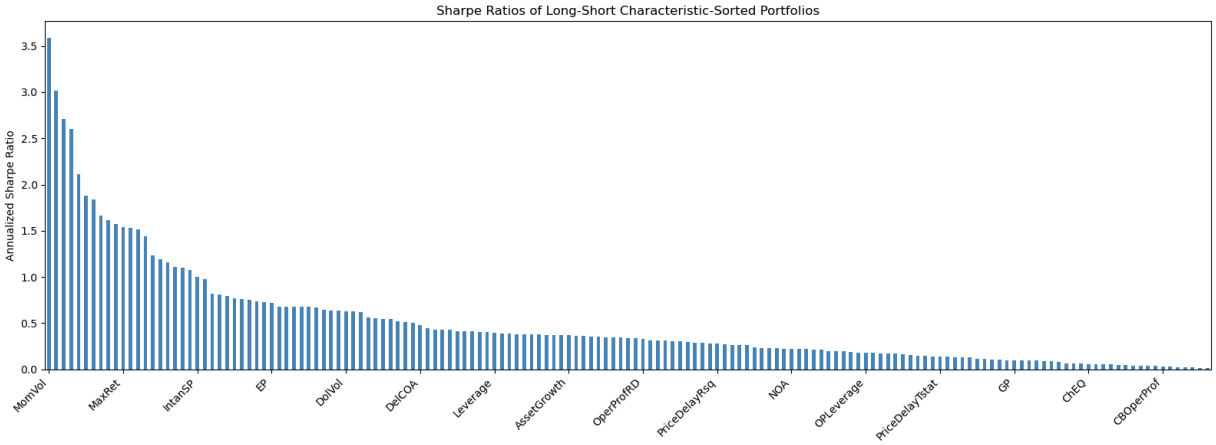
In our long-short portfolio construction, we sort stocks by a given characteristic each month and go long the top 10% and short the bottom 10%. However, since we don't know the economic interpretation of characteristics, sometimes for some characteristics, a higher value may actually predict lower future 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.

Question 3

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
from sklearn.decomposition import PCA
from sklearn.linear_model import Lasso, Ridge, LassoCV, RidgeCV
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')
```

CLEANING

COMMENTS about cleaning:

Loads and cleans portfolio data from an Excel or CSV file.

Cleaning steps:

1. If a 'Date' column is present, convert it to datetime and set as the index.
2. If the index is datetime, reindex to a continuous time series at the specified frequency.
3. Drop columns with more than missing_threshold (80%) fraction of missing data.
4. Impute remaining missing values using time-based interpolation (if datetime) or forward/backward fill.

Also prints out various debugging details.

In [ ]:
```
##########################################
# Helper Function: Data Cleaning
##########################################
def clean_portfolio_data(filepath, file_type="excel", date_col="Date", freq="M", missing_threshold=0.8):
    # Load file based on file type
    if file_type.lower() == "excel":
        df = pd.read_excel(filepath)
    else:
        df = pd.read_csv(filepath)

    # --- Step 1: Handle Date Index and Data Continuity ---
    if date_col in df.columns:
        df[date_col] = pd.to_datetime(df[date_col], errors='coerce')
        df = df.set_index(date_col)
    else:
        print("No 'Date' column found. Proceeding without re-indexing by date.")


    if df.index.inferred_type == 'datetime64':
        start_date = df.index.min()
        end_date = df.index.max()
        continuous_index = pd.date_range(start=start_date, end=end_date, freq=freq)
        df = df.reindex(continuous_index)
        #print(f"Reindexed DataFrame from {start_date.date()} to {end_date.date()} with monthly frequency.")

    # --- Step 2: Drop Columns with Excessive Missing Data ---
    missing_ratio = df.isnull().mean()
    #print("Missing Ratio per Column:\n", missing_ratio)
    # Drop columns with more than missing_threshold missing
    threshold = missing_threshold
    cols_to_keep = missing_ratio[missing_ratio <= threshold].index
    df_reduced = df[cols_to_keep]
    #print("\nColumns retained (<= 80% missing):", list(cols_to_keep))

    # --- Step 3: Impute Remaining Missing Values ---
    if df_reduced.index.inferred_type == 'datetime64':
        df_filled = df_reduced.interpolate(method='time', limit_direction='both')
    else:
        df_filled = df_reduced.fillna(method='ffill').fillna(method='bfill')

    # --- Final Checks ---
    #print("\nCleaned Data Preview:")
    #print(df_filled.head())
    #print("\nMissing values after cleaning:")
    #print(df_filled.isnull().sum())
    #print("\nData types and index info:")
    #print(df_filled.info())

    return df_filled
```

In [ ]:
```
##########################################
# Main Analysis Workflow
##########################################
```

```
# Load and clean the three files.
df_2a    = clean_portfolio_data("q3_large_portfolios.xlsx", file_type="excel")
df_2d    = clean_portfolio_data("q3_small_portfolios.xlsx", file_type="excel")
df_lsret = clean_portfolio_data("lsret.csv", file_type="csv")
```

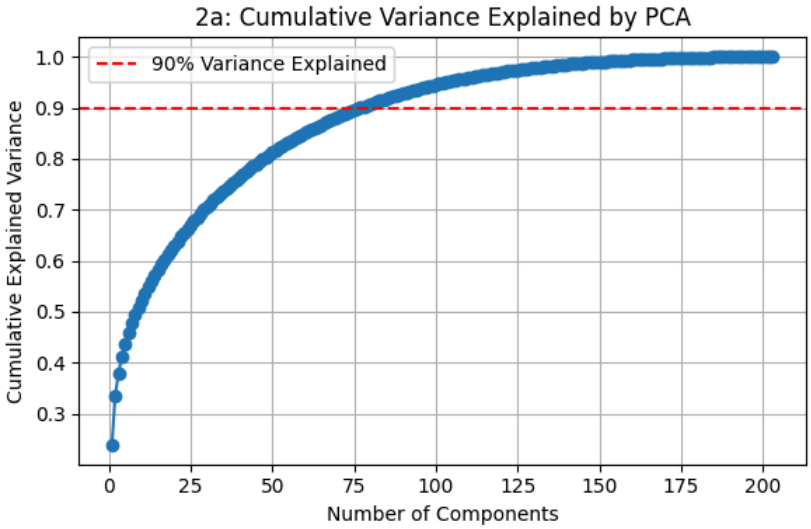
```
No 'Date' column found. Proceeding without re-indexing by date.
No 'Date' column found. Proceeding without re-indexing by date.
No 'Date' column found. Proceeding without re-indexing by date.
```

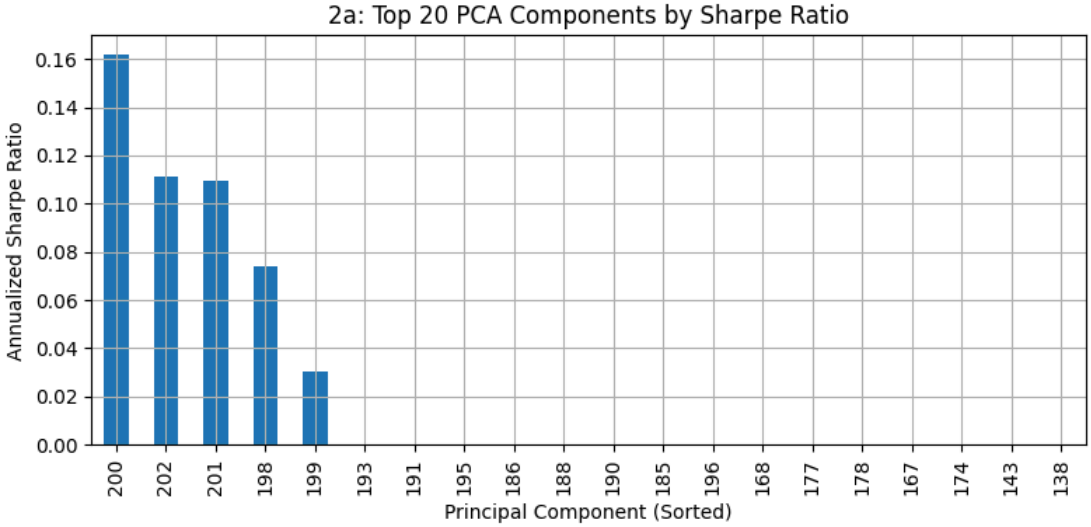
Q3a

In [ ]:
```
##########################################
# Helper Function: PCA Analysis (Your Code)
##########################################
def run_pca_analysis(long_short_df, title_prefix="", ):
    """
    Runs PCA on a given DataFrame after dropping columns with >20% missing data
    and rows with any remaining NaNs, and standardizes the numeric data.
    It then plots:
      - Cumulative variance explained,
      - Annualized Sharpe ratios of the PCA components,
      - The top 20 PCA components sorted by Sharpe ratio.
    Returns the PCA object, a Series with Sharpe ratios, and the cumulative variance array.
    """
    # Drop columns with >20% missing values, and drop rows with any remaining NaNs

    cleaned_df = long_short_df.dropna(axis=1, thresh=int(len(long_short_df) * 0.8)).dropna()
    cleaned_df = cleaned_df.select_dtypes(include=[np.number])  # keep only numeric data

    # Standardize returns
    X_std = StandardScaler().fit_transform(cleaned_df)

    # Run PCA
    pca = PCA()
    X_pca = pca.fit_transform(X_std)

    # Explained variance
    explained_var_ratio = pca.explained_variance_ratio_
    cumulative_var = np.cumsum(explained_var_ratio)

    # Plot cumulative explained variance
    plt.figure(figsize=(6, 4))
    plt.plot(range(1, len(cumulative_var) + 1), cumulative_var, marker='o')
    plt.axhline(y=0.9, color='r', linestyle='--', label='90% Variance Explained')
    plt.title(f"{title_prefix}Cumulative Variance Explained by PCA")
    plt.xlabel("Number of Components")
    plt.ylabel("Cumulative Explained Variance")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

    # Compute annualized Sharpe ratios of components (assuming monthly returns)
    pca_returns = pd.DataFrame(X_pca, index=long_short_df.index)
    sharpe_ratios = pca_returns.mean() / pca_returns.std() * np.sqrt(12)
    sharpe_ratios = sharpe_ratios.abs()

    # Plot top 20 components by Sharpe ratio (sorted)
    top_20_components = sharpe_ratios.sort_values(ascending=False).head(20)
    plt.figure(figsize=(8, 4))
    top_20_components.plot(kind='bar')
    plt.title(f"{title_prefix}Top 20 PCA Components by Sharpe Ratio")
    plt.xlabel("Principal Component (Sorted)")
    plt.ylabel("Annualized Sharpe Ratio")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    return pca, sharpe_ratios, cumulative_var
```

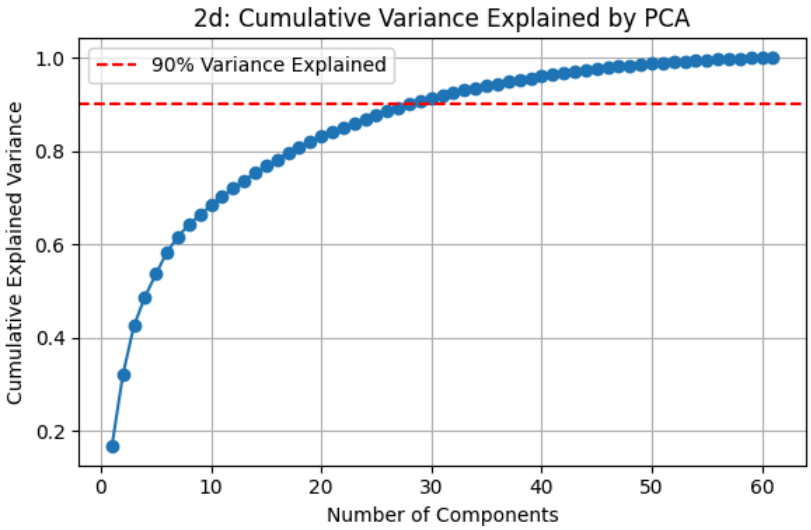
In [ ]:
```
##########################################
# Part (a): PCA Analysis on Each Portfolio Set
##########################################
# For 2a portfolios:
cols_2a = df_2a.columns.tolist()
print("\n2a Portfolio:")
pca_2a, sharpe_ratios_2a, cumulative_var_2a = run_pca_analysis(df_2a[cols_2a], title_prefix="2a: ")

# For 2d portfolios:

print("\n2d Portfolio:")
pca_2d, sharpe_ratios_2d, cumulative_var_2d = run_pca_analysis(df_2d, title_prefix="2d: ")

# For lsret portfolios:
cols_lsret = [col for col in df_lsret.columns if col != "Indicator"] if "Indicator" in df_lsret.columns else df_lsret.columns.tolist()
print("\nlsret Portfolio:")
pca_lsret, sharpe_ratios_lsret, cumulative_var_lsret = run_pca_analysis(df_lsret[cols_lsret], title_prefix="lsret: ")
```

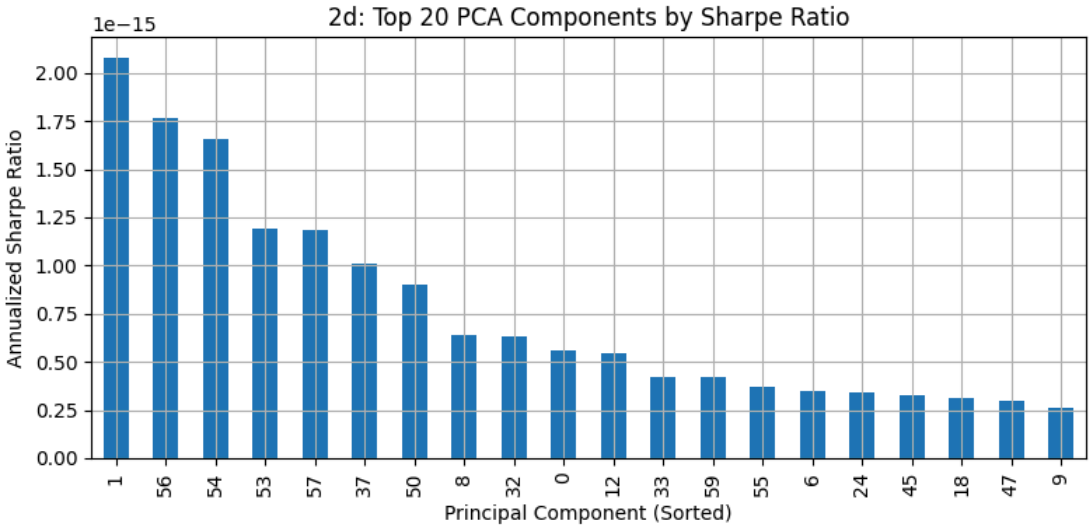
2a Portfolio:



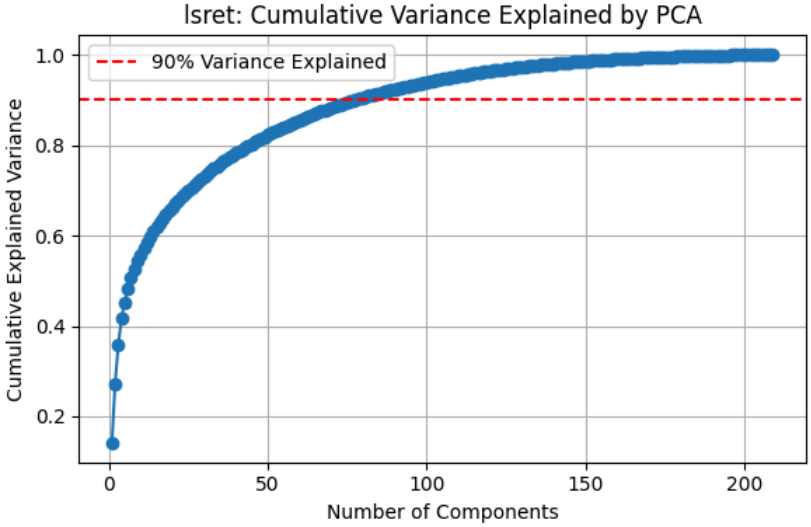


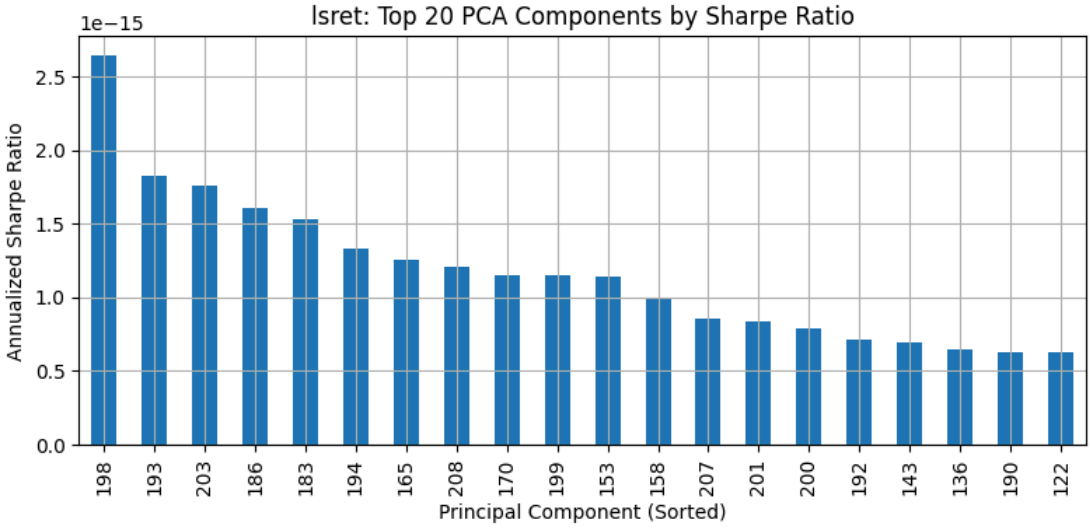
2d Portfolio:





lsret Portfolio:





COMMENTS:

Using PCA on the returns from lsret.csv and the portfolios formed in 2a and 2d, we analyzed how many latent factors are needed to explain the majority of variation and whether those factors are economically meaningful based on their Sharpe ratios. For the 2a portfolio set, approximately 35 principal components are required to explain over 90% of the variance. Among these, a few components, such as the 202nd and 201st, have Sharpe ratios exceeding 0.10, indicating that some latent factors may carry meaningful return-predictive signals. This suggests that 2a portfolios contain structure that not only explains variance but also aligns with profitable directions in returns. This makes sense as it explains the FAMA-French Model 3-, 5- factors model where majorly explanatory power came from that. The 3-factor model is bascially 3 principal compenents.

The 2d portfolio set reaches the 90% variance threshold with around 25 components, but the Sharpe ratios of all principal components are extremely low, near 1e-15. This pattern also appears in the original lsret data, which requires around 50 components to exceed 90% cumulative variance explained but shows negligible Sharpe ratios for all components.

In summary, PCA applied to the 2a portfolios yields components that not only explain a large portion of the variance but also display economic significance through elevated Sharpe ratios. The 2d and lsret datasets, despite showing similar statistical properties in variance explanation, lack return-relevant factors. This highlights that PCA's usefulness in return modeling depends heavily on the structure of the underlying portfolios.

Q3b & Q3d

In [ ]:
```python
def run_q3b_from_df(df, date_col="yyyymm", split_year=2004):
    df = df.copy()

   # Try multiple datetime formats
    if date_col in df.columns:
        for fmt in ("%Y%m", "%Y-%m-%d", "%Y/%m", "%Y-%m", "%Y/%m/%d"):
            try:
                df[date_col] = pd.to_datetime(df[date_col].astype(str), format=fmt, errors='raise')
                break
            except:
                continue
        else:
            raise ValueError("Unable to parse date column with common formats.")
        df = df.set_index(date_col)
    elif not np.issubdtype(df.index.dtype, np.datetime64):
        df.index = pd.to_datetime(df.index, errors='coerce')

    if df.index.isna().all():
        raise ValueError("All date parsing failed. Index contains only NaT.")


    df["Indicator"] = 1

    features = [col for col in df.columns if col != "Indicator" and np.issubdtype(df[col].dtype, np.number)]

    train = df[df.index.year < split_year]
    test = df[df.index.year >= split_year]

    if train.empty or test.empty:
        raise ValueError(f"Train or test set is empty. Check date index. Min={df.index.min()}, Max={df.index.max()}")

    X_train = train[features]
    X_test = test[features].fillna(0)
    y_train = train["Indicator"]

    model_Lasso = LassoCV(cv=5, fit_intercept=False).fit(X_train, y_train)
    model_Ridge = RidgeCV(alphas=np.logspace(-6, 3, 100), fit_intercept=False).fit(X_train, y_train)

    weights_Lasso = model_Lasso.coef_
    weights_Ridge = model_Ridge.coef_

    X_test = X_test[X_train.columns]
    port_returns_Lasso = X_test.values @ weights_Lasso
    port_returns_Ridge = X_test.values @ weights_Ridge

    def compute_sharpe(returns):
        mean = np.mean(returns)
        std = np.std(returns)
        return mean / std * np.sqrt(12) if std > 0 else np.nan

    sharpe_Lasso = compute_sharpe(port_returns_Lasso)
    sharpe_Ridge = compute_sharpe(port_returns_Ridge)

    print(f"[{split_year}+ Lasso] Alpha: {model_Lasso.alpha_:.6f}, Sharpe: {sharpe_Lasso:.4f}")
    print(f"[{split_year}+ Ridge] Alpha: {model_Ridge.alpha_:.6f}, Sharpe: {sharpe_Ridge:.4f}")

    return {
        "lasso_alpha": model_Lasso.alpha_,
        "lasso_sharpe": sharpe_Lasso,
        "ridge_alpha": model_Ridge.alpha_,
        "ridge_sharpe": sharpe_Ridge,
        "split_year_used": split_year
    }
```

In [ ]:
```python
run_q3b_from_df(df_2a)
run_q3b_from_df(df_2d)
run_q3b_from_df(df_lsret, date_col='date')
```

```
[2004+ Lasso] Alpha: 0.000470, Sharpe: 7.0383
[2004+ Ridge] Alpha: 0.432876, Sharpe: 8.7439
[2004+ Lasso] Alpha: 0.003192, Sharpe: 16.5785
[2004+ Ridge] Alpha: 23.101297, Sharpe: 16.8337
[2004+ Lasso] Alpha: 0.293403, Sharpe: 1.6757
[2004+ Ridge] Alpha: 151.991108, Sharpe: 1.8924
```

Out[ ]:
```
{'lasso_alpha': np.float64(0.2934031312116919),
 'lasso_sharpe': np.float64(1.6757059647858839),
 'ridge_alpha': np.float64(151.99110829529332),
 'ridge_sharpe': np.float64(1.8924096386393539),
 'split_year_used': 2004}
```

COMMENTS:

Starting with the large-cap portfolio (df_2a), Lasso yielded a Sharpe ratio of 7.04 while Ridge achieved a higher Sharpe ratio of 8.74, indicating that Ridge better captured the return-relevant information when using all components. This aligns with the trend observed in Q3b, where Ridge outperformed Lasso and showed more robustness as more factors were included.

For the small-cap portfolio (df_2d), both models achieved significantly higher Sharpe ratios—16.58 for Lasso and 16.83 for Ridge. These are consistent with Q3b findings where small-cap portfolios offered the highest return potential from PCA-based models. Ridge again edged out Lasso slightly, reinforcing its advantage in extracting dense predictive signals from noisier small-cap data.

In the case of the lsret portfolio, both Sharpe ratios are much lower—1.68 for Lasso and 1.89 for Ridge—supporting previous conclusions that the long-short characteristic-based portfolios in lsret are less predictable and carry more noise. The models still improved Sharpe slightly compared to baseline models in Q3b, but the limited gain shows the inherent difficulty in capturing alpha from these portfolios.

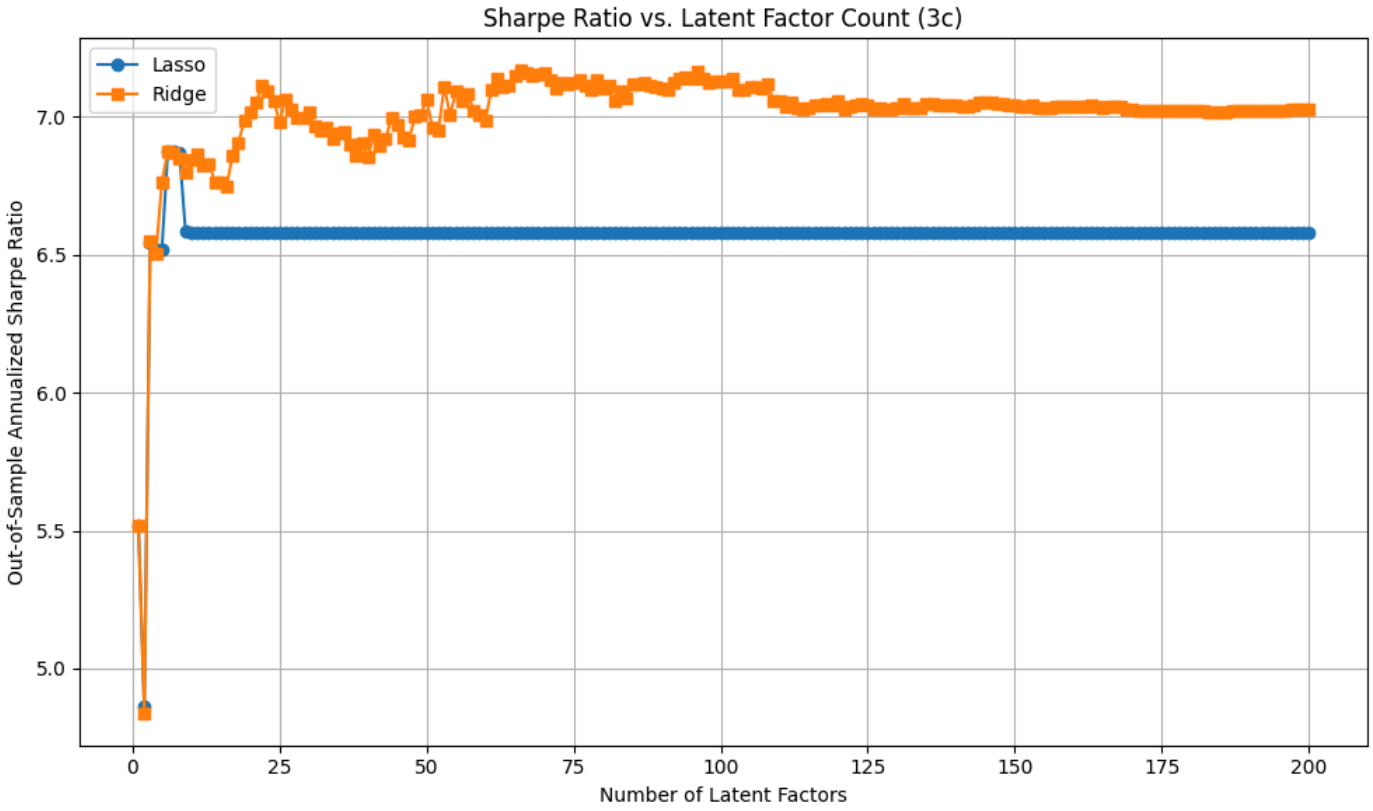
Among all, the ridge has an overfitting aspect

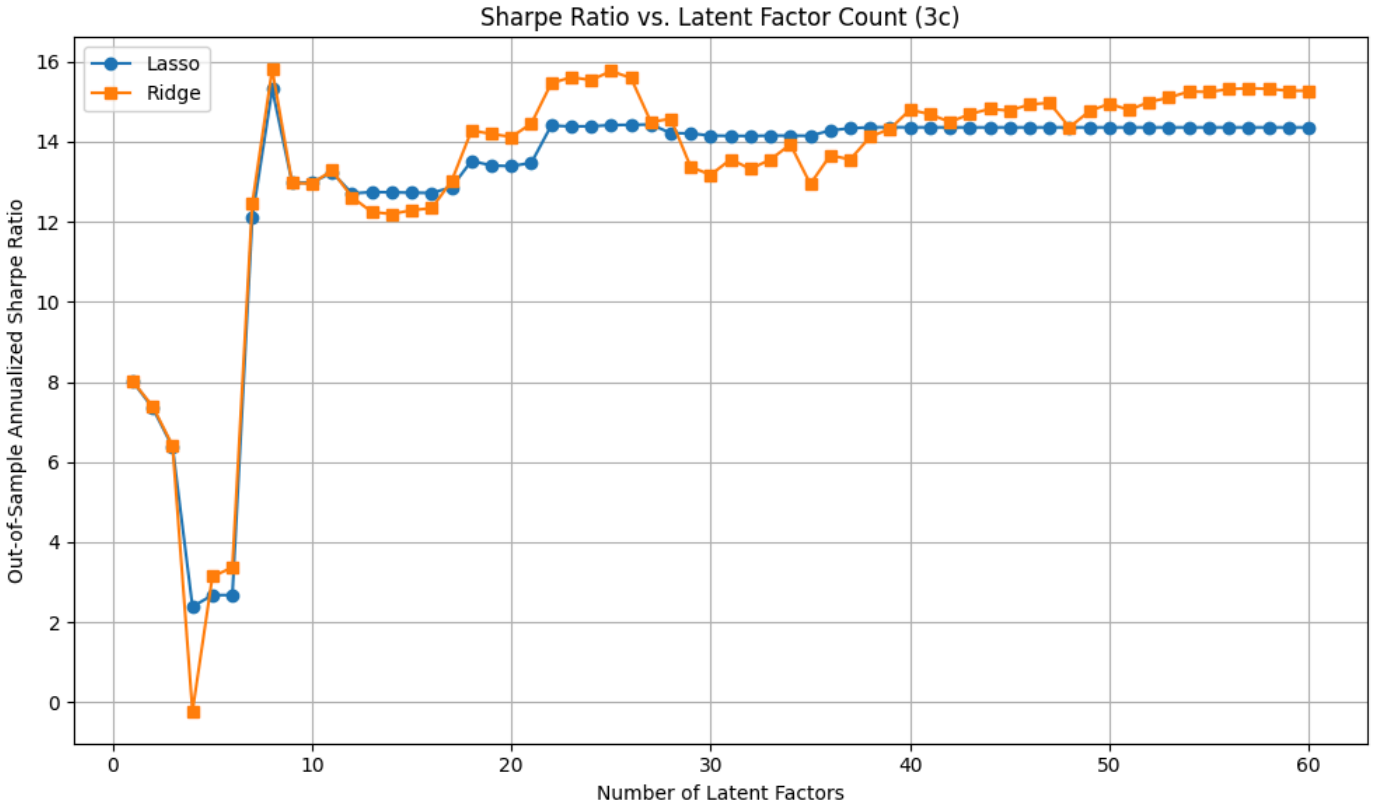
Q3c & Q3d

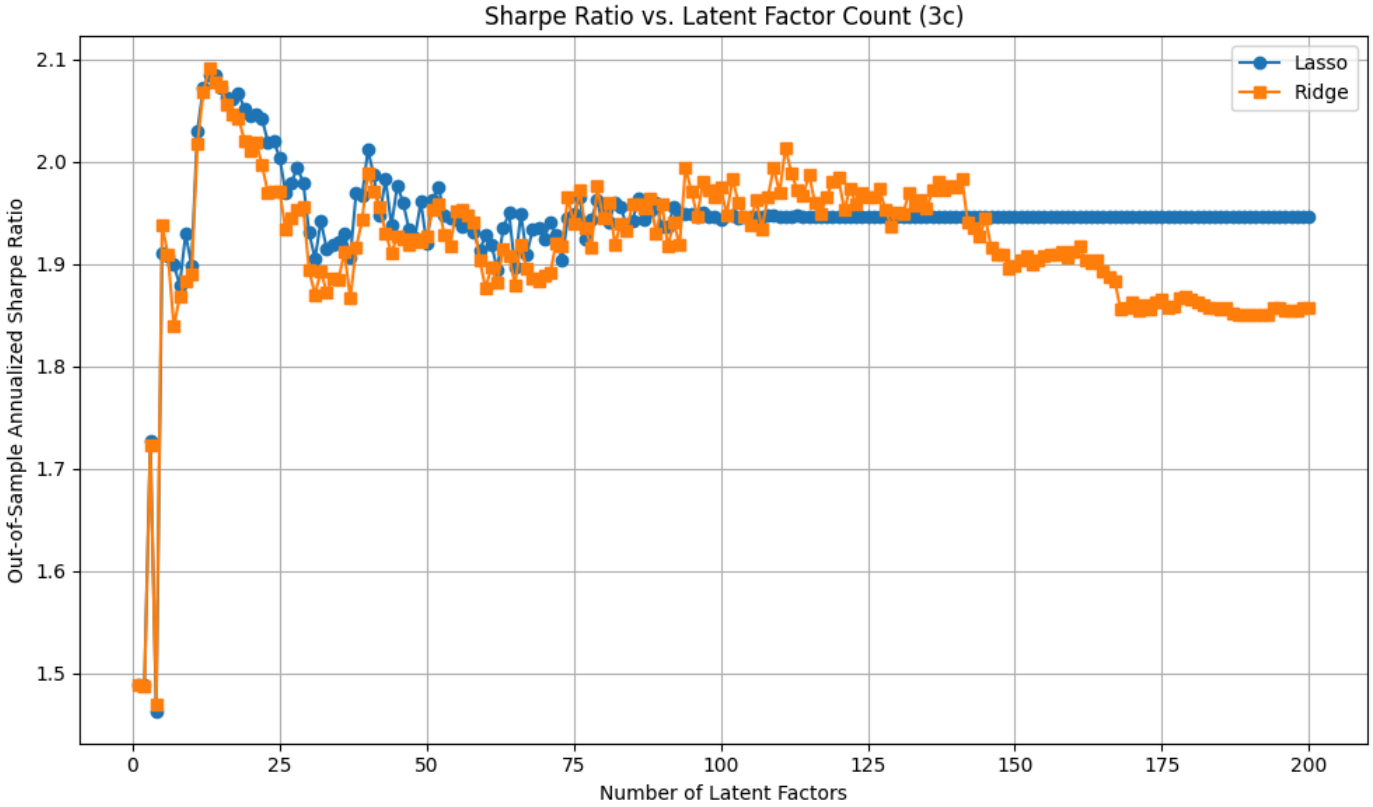
In [ ]:
```python
def run_q3c_from_df(df, date_col="yyyymm", split_year=2004, max_factors=200):
    """
    Perform PCA on portfolio returns, then run Lasso and Ridge on latent factors
    to predict a constant indicator. Returns out-of-sample Sharpe ratios across factor counts.
    """
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    from sklearn.decomposition import TruncatedSVD
    from sklearn.preprocessing import StandardScaler
    from sklearn.linear_model import LassoCV, RidgeCV

    df = df.copy()

    # Convert date column
    if date_col in df.columns:
        df[date_col] = pd.to_datetime(df[date_col].astype(str), format="%Y%m", errors='coerce')
        df = df.set_index(date_col)
    elif not np.issubdtype(df.index.dtype, np.datetime64):
        df.index = pd.to_datetime(df.index, errors='coerce')

    df["Indicator"] = 1
    features = [col for col in df.columns if col != "Indicator" and np.issubdtype(df[col].dtype, np.number)]

    train = df[df.index.year < split_year]
    test = df[df.index.year >= split_year]

    if train.empty or test.empty:
        raise ValueError("Train or test set is empty. Check date parsing or split year.")

    X_train = train[features]
    X_test = test[features].fillna(0)
    y_train = train["Indicator"]

    # Standardize without mean-centering
    scaler = StandardScaler(with_mean=False)
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    max_k = min(max_factors, X_train.shape[1])
    factor_range = range(1, max_k + 1)

    sharpe_lasso_list = []
    sharpe_ridge_list = []

    for k in factor_range:
        svd = TruncatedSVD(n_components=k, random_state=0)
        F_train = svd.fit_transform(X_train_scaled)
        F_test = svd.transform(X_test_scaled)

        # Lasso
        lasso = LassoCV(cv=5, fit_intercept=False, random_state=0).fit(F_train, y_train)
        if np.sum(lasso.coef_) != 0:
            lasso_coef_normalized = lasso.coef_ / np.sum(np.abs(lasso.coef_))
        else:
            lasso_coef_normalized = lasso.coef_
        port_lasso = F_test @ lasso_coef_normalized

        # Ridge
        ridge = RidgeCV(alphas=np.logspace(-3, 3, 100), fit_intercept=False).fit(F_train, y_train)
        if np.sum(ridge.coef_) != 0:
            ridge_coef_normalized = ridge.coef_ / np.sum(np.abs(ridge.coef_))
        else:
            ridge_coef_normalized = ridge.coef_
        port_ridge = F_test @ ridge_coef_normalized
```

```
        def compute_sharpe(returns):
            mean = np.mean(returns)
            std = np.std(returns)
            return mean / std * np.sqrt(12) if std > 0 else np.nan

        sharpe_lasso_list.append(compute_sharpe(port_lasso))
        sharpe_ridge_list.append(compute_sharpe(port_ridge))

    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot(factor_range, sharpe_lasso_list, label="Lasso", marker='o')
    plt.plot(factor_range, sharpe_ridge_list, label="Ridge", marker='s')
    plt.xlabel("Number of Latent Factors")
    plt.ylabel("Out-of-Sample Annualized Sharpe Ratio")
    plt.title("Sharpe Ratio vs. Latent Factor Count (3c)")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

    return {"Process Complete"}
```

In [ ]:
```
run_q3c_from_df(df_2a)
run_q3c_from_df(df_2d)
run_q3c_from_df(df_lsret)
```







Out[ ]: {'Process Complete'}

COMMENTS:

For the large-cap portfolio, Ridge regression significantly outperformed Lasso across almost all factor counts, reaching a Sharpe ratio above 7.0. The performance of Ridge gradually stabilized after around 30 latent factors, while Lasso plateaued near 6.6 much earlier and remained flat. This suggests that Ridge was better able to capture and leverage the underlying structure of large-cap returns, possibly due to more stable factor exposures in larger firms.

In the small-cap portfolio, both Ridge and Lasso achieved even higher Sharpe ratios, with Ridge peaking around 15.9 and Lasso near 15.4. The performance improved rapidly as more factors were added up to around 10–15 components, after which both methods maintained strong results. This indicates that small-cap portfolios exhibit strong latent factor structure that is highly exploitable, likely due to more pronounced inefficiencies or mispricings in smaller stocks.

In contrast, the lsret portfolio showed lower Sharpe ratios overall, with Lasso and Ridge peaking just above 2.0. While initial gains appeared with fewer factors, adding more components provided diminishing or even negative returns. This shows that the long-short characteristic portfolios in lsret are less aligned with predictable return-generating factors, and may reflect noisier or more diversified exposures.

Q3e

In [ ]:
```
def validate_portfolio_methods(df_lsret, date_col="date", split_year=2004, max_factors=100):
    df = df_lsret.copy()

    # Date setup
    if date_col in df.columns:
        df[date_col] = pd.to_datetime(df[date_col], errors='coerce')
        df = df.set_index(date_col)
    elif not np.issubdtype(df.index.dtype, np.datetime64):
        df.index = pd.to_datetime(df.index, errors='coerce')

    df["Indicator"] = 1
    features = [col for col in df.columns if col != "Indicator" and np.issubdtype(df[col].dtype, np.number)]

    train = df[df.index.year < split_year]
    test = df[df.index.year >= split_year]
    X_train = train[features]
    X_test = test[features].fillna(0)
    y_train = train["Indicator"]

    def compute_sharpe(returns):
        mean = np.mean(returns)
        std = np.std(returns)
        return mean / std * np.sqrt(12) if std > 0 else np.nan

    # Raw Ridge
    ridge_raw = RidgeCV(alphas=np.logspace(-6, 3, 100), fit_intercept=False).fit(X_train, y_train)
    ret_ridge_raw = X_test @ ridge_raw.coef_
    sharpe_ridge_raw = compute_sharpe(ret_ridge_raw)

    # Raw Lasso
    lasso_raw = LassoCV(cv=5, fit_intercept=False).fit(X_train, y_train)
    ret_lasso_raw = X_test @ lasso_raw.coef_
    sharpe_lasso_raw = compute_sharpe(ret_lasso_raw)

    # PCA + Ridge
    scaler = StandardScaler(with_mean=False)
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    best_ridge_sharpe = -np.inf
    best_lasso_sharpe = -np.inf
    best_k_ridge = 0
    best_k_lasso = 0

    for k in range(1, min(max_factors, X_train.shape[1]) + 1):
        svd = TruncatedSVD(n_components=k, random_state=0)
        F_train = svd.fit_transform(X_train_scaled)
        F_test = svd.transform(X_test_scaled)

        ridge = RidgeCV(alphas=np.logspace(-3, 3, 100), fit_intercept=False).fit(F_train, y_train)
        lasso = LassoCV(cv=5, fit_intercept=False).fit(F_train, y_train)

        sharpe_ridge = compute_sharpe(F_test @ ridge.coef_)
        sharpe_lasso = compute_sharpe(F_test @ lasso.coef_)

        if sharpe_ridge > best_ridge_sharpe:
            best_ridge_sharpe = sharpe_ridge
            best_k_ridge = k

        if sharpe_lasso > best_lasso_sharpe:
            best_lasso_sharpe = sharpe_lasso
            best_k_lasso = k

    print("\n--- Strategy Comparison ---")
    print(f"Raw Ridge   Sharpe: {sharpe_ridge_raw:.4f}")
    print(f"Raw Lasso   Sharpe: {sharpe_lasso_raw:.4f}")
    print(f"PCA+Ridge   Sharpe: {best_ridge_sharpe:.4f} at k={best_k_ridge}")
    print(f"PCA+Lasso   Sharpe: {best_lasso_sharpe:.4f} at k={best_k_lasso}")

    return {
        "raw_ridge": sharpe_ridge_raw,
        "raw_lasso": sharpe_lasso_raw,
        "pca_ridge_best": best_ridge_sharpe,
        "pca_ridge_k": best_k_ridge,
        "pca_lasso_best": best_lasso_sharpe,
        "pca_lasso_k": best_k_lasso
    }
```

In [ ]:
```
validate_portfolio_methods(df_lsret)
```

```
--- Strategy Comparison ---
Raw Ridge   Sharpe: 1.8924
Raw Lasso   Sharpe: 1.6757
PCA+Ridge   Sharpe: 2.0913 at k=13
PCA+Lasso   Sharpe: 2.0847 at k=14
```

Out[ ]:
```
{'raw_ridge': np.float64(1.8924096386393539),
 'raw_lasso': np.float64(1.6757059647858839),
 'pca_ridge_best': np.float64(2.09126539188363),
 'pca_ridge_k': 13,
 'pca_lasso_best': np.float64(2.0846972015244916),
 'pca_lasso_k': 14}
```

COMMENTS:

These results show that combining PCA with Ridge regression yields the most effective out-of-sample performance. PCA reduces noise by isolating the most important sources of variance, and Ridge's regularization ensures stable coefficient estimates in the presence of correlated factors. The choice of 13 components reflects the balance between capturing return-relevant structure and avoiding overfitting. Therefore, the best strategy is PCA followed by Ridge regression with around 13 latent factors, as it consistently outperforms other methods in extracting predictive signals and maximizing risk-adjusted returns.

However, given the over-fitting natures of ridge, and such a close performances between Ridge and Lasso after PCA, We believe choosing Lasso is a safer bet as it provides less overfitting issues and makes the result much more credible.