Machine Learning for Economics and Finance

Problem Set 2

Ole Wilms

July 29, 2024

Important Instructions

- In this problem set you are asked to apply the machine learning techniques we covered in the past weeks
- In case you struggle with some problems, please post your questions on the OpenOlat discussion board.
- We will discuss the solutions for the problem set on `MONTH DAY`

Setup

Assume the same setup as in *Problem Set 1* but now you try to improve the return predictions using the machine learning approaches we have discussed in class. For this you are asked to use the same training and test datasets we constructed in *Problem Set 1*.

Preliminaries

[1]:
```
# Loading needed packages for this ProblemSet2
import pyreadr
import pandas as pd
import numpy as np
import statsmodels.api as sm
from statsmodels.formula.api import ols
from sklearn.linear_model import LinearRegression, LogisticRegression, Ridge,␣
 ↪Lasso, RidgeCV, LassoCV
from sklearn.model_selection import cross_val_score, KFold, GridSearchCV
from sklearn.metrics import r2_score, mean_squared_error, accuracy_score
from matplotlib.pyplot import subplots
from statsmodels.api import OLS
import sklearn.model_selection as skm
import sklearn.linear_model as skl
from sklearn.preprocessing import StandardScaler
from functools import partial
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from sklearn.model_selection import train_test_split
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import TransformedTargetRegressor
from sklearn.pipeline import make_pipeline
from sklearn.metrics import PredictionErrorDisplay, median_absolute_error
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import cross_validate

# Setup options for the following matplotlib plots:
plt.rcParams.update({
    'figure.figsize': (8, 6),
    'font.family': 'serif',
    'font.size': 12,
    'axes.titlesize': 14,
    'axes.grid': False,
    'lines.linewidth': 2
})
```

[2]:
```
# Load and prepare data (your existing code)
df = pyreadr.read_r('~/Desktop/stockmarketdata.rds')
df = df[None]
dfRAW = df.copy(deep=True)

# Lead returns (shift by -1)
```

```
df['ret'] = df['ret'].shift(-1)

# Remove missing values
df = df.dropna()
```

Splitting the Dataset into train- and testdata

[3]:
```
# Split data into train and test sample
df['date'] = df['date'].astype(np.int64)
split_date = 19944
split_ind = df.index[df['date'] == split_date][0]

train_data = df.loc[:split_ind]
train_data = train_data.drop('date', axis=1)
test_data = df.loc[split_ind + 1:]
test_data = test_data.drop('date', axis=1)

# Prepare X and y
X_train = train_data.drop(columns=['ret'])
y_train = train_data['ret']
X_test = test_data.drop(columns=['ret'])
y_test = test_data['ret']
```

Exploration and visualization of the dataset (additional)

[5]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 272 entries, 92 to 363
Data columns (total 8 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   date    272 non-null    int64
 1   ret     272 non-null    float64
 2   DP      272 non-null    float64
 3   CS      272 non-null    float64
 4   ntis    272 non-null    float64
 5   cay     272 non-null    float64
 6   TS      272 non-null    float64
 7   svar    272 non-null    float64
dtypes: float64(7), int64(1)
memory usage: 27.2 KB
```

[7]:
```
# Descriptive Statistics of the Data
df.describe().T
```

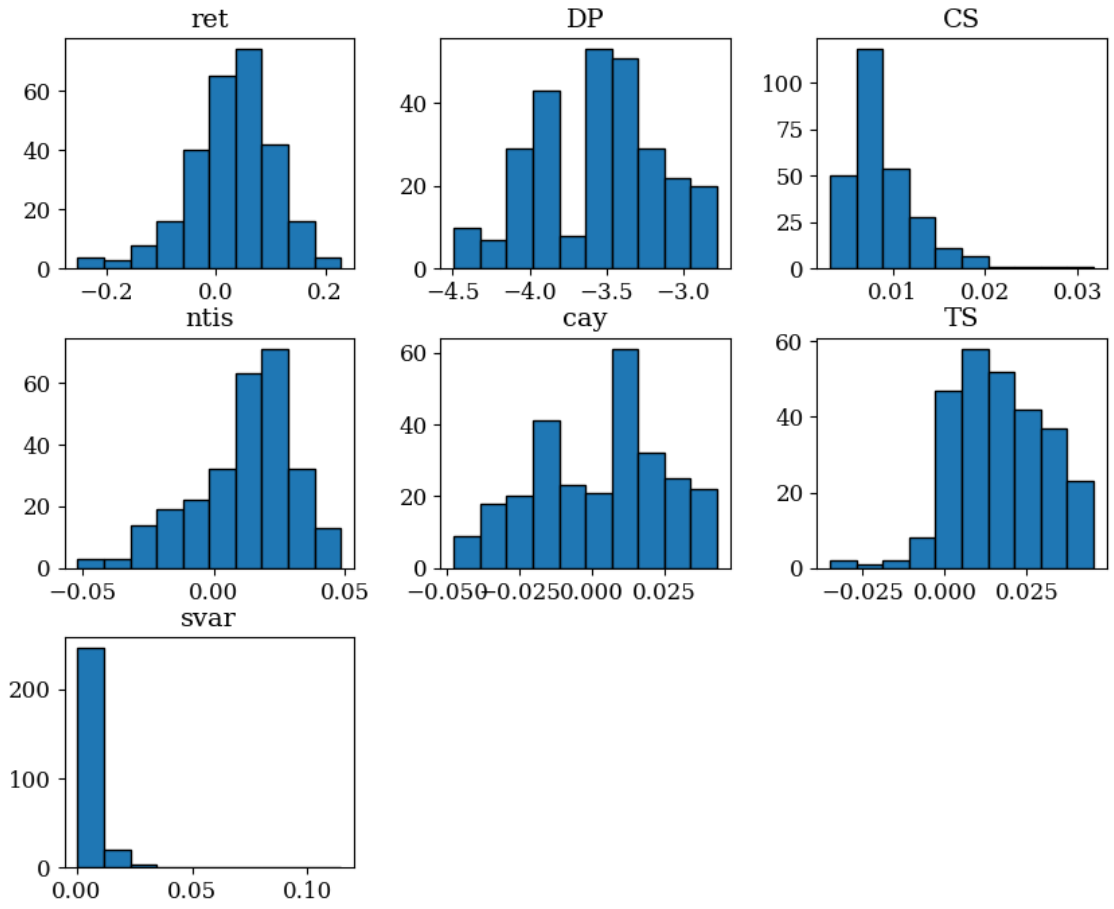
[7]:

	count	mean	std	min	25%	\
date	272.0	19857.500000	196.642857	19521.000000	19689.250000	

```
ret   272.0      0.028608    0.078468     -0.252154     -0.016725
DP    272.0     -3.553095    0.404929     -4.493159     -3.906620
CS    272.0      0.009002    0.003875      0.003243      0.006465
ntis  272.0      0.011831    0.019359     -0.051831      0.001356
cay   272.0      0.002189    0.022592     -0.047607     -0.016987
TS    272.0      0.016710    0.013993     -0.035000      0.006725
svar  272.0      0.005822    0.009721      0.000370      0.002142

                50%           75%           max
date  19857.500000  20025.750000  20194.000000
ret       0.035821      0.076914      0.228211
DP       -3.498891     -3.279654     -2.778536
CS        0.008044      0.010598      0.031668
ntis      0.015415      0.025945      0.048391
cay       0.007717      0.018797      0.042897
TS        0.016000      0.027325      0.045300
svar      0.003476      0.005951      0.114436
```

[8]:
```
df.loc[:, df.columns != "date"].hist(figsize=(10, 8), edgecolor='black',␣
 ↪grid=False)
# Graphical Overview of the Distributions of Individual Variables in the Data
# The variable "date" is not explicitly excluded here.
```

[8]:
```
array([[<Axes: title={'center': 'ret'}>, <Axes: title={'center': 'DP'}>,
        <Axes: title={'center': 'CS'}>],
       [<Axes: title={'center': 'ntis'}>,
        <Axes: title={'center': 'cay'}>, <Axes: title={'center': 'TS'}>],
       [<Axes: title={'center': 'svar'}>, <Axes: >, <Axes: >]],
      dtype=object)
```



Linear Model

[9]:
```
# Train linear model for comparison (your existing code)
model_all = LinearRegression()
model_all.fit(X_train, y_train)
y_pred_train_lm = model_all.predict(X_train)
mse_train_lm = mean_squared_error(y_train, y_pred_train_lm)
y_pred_test_lm = model_all.predict(X_test)
mse_test_lm = mean_squared_error(y_test, y_pred_test_lm)

# print("LINEAR MODEL RESULTS:")
print(f"In-sample MSE:     {mse_train_lm:.5f}")
print(f"Out-of-sample MSE: {mse_test_lm:.5f}")
```

```
In-sample MSE:     0.00505
Out-of-sample MSE: 0.00861
```

Question 1: Shrinkage Methods

1.1 Ridge Regression

Fit a ridge regression using the training data. Determine the optimal penalty parameter λ using 5-fold cross validation (set the seed to 2 before you run the CV). Provide a plot of the cross-validation MSE as a function of $\log(\lambda)$ and interpret the outome.

[10]:
```
# Set up Ridge regression with cross-validation
np.random.seed(2)  # Set seed for reproducibility

# Define lambda grid (alpha in sklearn)
lambda_grid = np.logspace(-6, 2, 100)  # From log(-6) to log(2)

# Perform Ridge regression with 5-fold cross-validation
# Note: store_cv_results only works with cv=None, so we'll use manual CV for␣
 ↪plotting
ridge_cv = RidgeCV(alphas=lambda_grid, cv=5, scoring='neg_mean_squared_error')
ridge_cv.fit(X_train, y_train)

# Get best lambda (alpha in sklearn)
best_lambda_ridge = ridge_cv.alpha_
print(f"Best lambda (Ridge): {best_lambda_ridge:.6f}")
print(f"Log of best lambda: {np.log(best_lambda_ridge):.4f}")
```

```
Best lambda (Ridge): 0.040370
Log of best lambda: -3.2097
```

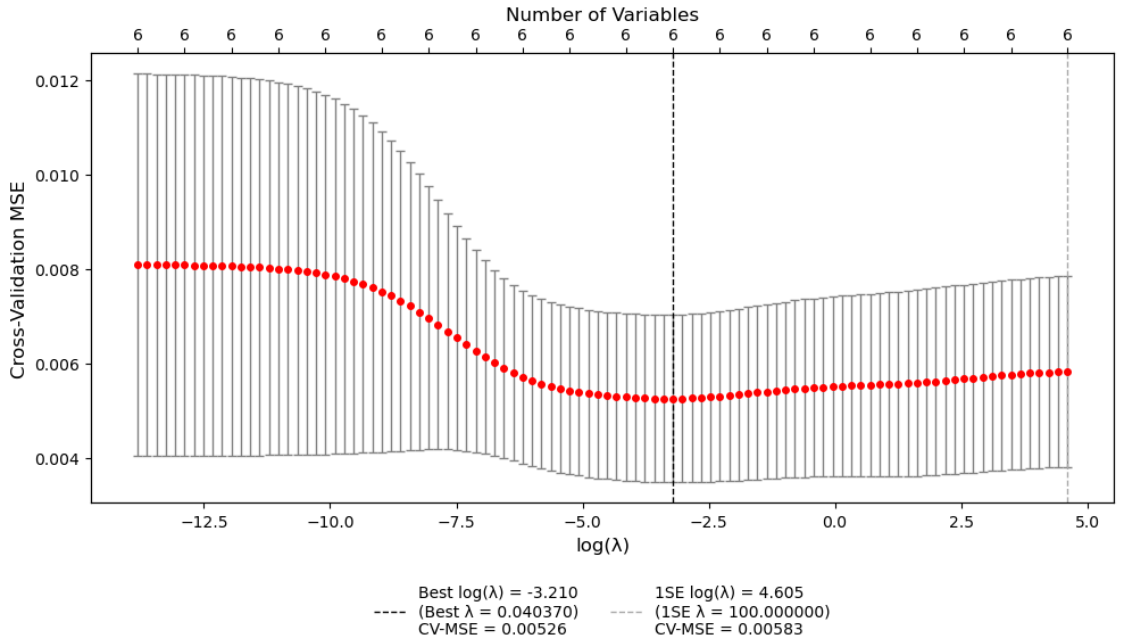
[13]:
```
from sklearn.model_selection import cross_validate

# Manual cross-validation to get detailed results for plotting (like R's cv.
 ↪glmnet)
print("Performing detailed cross-validation for plotting...")

cv_mse_mean = []
cv_mse_std = []

for alpha in lambda_grid:
  ridge_temp = Ridge(alpha=alpha)
  cv_results = cross_validate(ridge_temp, X_train, y_train,
                              cv=5, scoring='neg_mean_squared_error',
                              return_train_score=False)
  cv_scores = -cv_results['test_score']  # Convert to positive MSE
  cv_mse_mean.append(cv_scores.mean())
  cv_mse_std.append(cv_scores.std())

cv_mse_mean = np.array(cv_mse_mean)
cv_mse_std = np.array(cv_mse_std)
```

```
# Index von best_lambda_ridge im Grid finden:
idx_best = np.where(lambda_grid == best_lambda_ridge)[0][0]
best_mse = cv_mse_mean[idx_best]

# Index für _1SE finden:
# Bestes MSE + 1SE-Grenze
threshold = best_mse + cv_mse_std[idx_best]

# Kandidaten-Lambdas finden, die <= threshold sind
candidates = np.where(cv_mse_mean <= threshold)[0]

# Nimm das größte  (also den einfachsten / regularisiertesten Kandidaten)
idx_1se = candidates[-1]
best_lambda_1se = lambda_grid[idx_1se]
best_mse_1se = cv_mse_mean[idx_1se]

# --- NEUER CODE FÜR y2-ACHSE: Anzahl Variablen berechnen ---
n_nonzero = []
for alpha in lambda_grid:
    ridge_temp = Ridge(alpha=alpha)
    ridge_temp.fit(X_train, y_train)
    # Ridge behält alle Variablen, aber wir können trotzdem die Anzahl Features␣
 ↪anzeigen
    n_nonzero.append(X_train.shape[1])  # Für Ridge immer alle Features
    # Alternative: Zeige "effective" number of parameters basierend auf␣
 ↪Koeffizienten-Größe
    # n_nonzero.append(np.sum(np.abs(ridge_temp.coef_) > np.max(np.
 ↪abs(ridge_temp.coef_)) * 0.01))

plt.style.use('default')
# --- GEÄNDERT: fig, ax1 für subplot mit y2-Achse ---
fig, ax1 = plt.subplots(figsize=(10, 6))

ax1.errorbar(np.log(lambda_grid),
             cv_mse_mean,
             yerr=cv_mse_std,
             capsize=3,
             color="red",
             ecolor="grey",
             elinewidth=1,
             fmt='o',
             markersize=4,
             )

ax1.axvline(np.log(best_lambda_ridge),
             color='black',
             linestyle='--',
```

```
            linewidth=1,
            label=(f'Best log( ) = {np.log(best_lambda_ridge):.3f}\n'
                   f'(Best   = {best_lambda_ridge:.6f})\n'
                   f'CV-MSE = {best_mse:.5f}'
                  )
            )

ax1.axvline(np.log(best_lambda_1se),
            color='darkgrey',
            linestyle='--',
            linewidth=1,
            label=(f'1SE log( ) = {np.log(best_lambda_1se):.3f}\n'
                   f'(1SE   = {best_lambda_1se:.6f})\n'
                   f'CV-MSE = {best_mse_1se:.5f}'
                  )
            )

ax1.set_xlabel('log( )', fontsize=12)
ax1.set_ylabel('Cross-Validation MSE', fontsize=12)
# ax1.set_title('Ridge Regression: Cross-Validation MSE vs log( )')
ax1.grid(False)

# --- Legende unter dem Plot, zentriert, nebeneinander ---
ax1.legend(bbox_to_anchor=(0.5, -0.15),
          loc='upper center',
          ncol=3,
          frameon=True,
          framealpha=0   # 0 = unsichtbar, 1 = voll sichtbar
          )

# --- NEUER CODE: y2-Achse oben für Anzahl Variablen ---
ax2 = ax1.twiny()
ax2.set_xlim(ax1.get_xlim())

# Ticks und Labels für die Anzahl der Variablen setzen
log_lambdas = np.log(lambda_grid)
# Zeige max 20 Ticks um Überlappung zu vermeiden
n_ticks = min(20, len(lambda_grid))
tick_indices = np.linspace(0, len(lambda_grid)-1, n_ticks, dtype=int)

ax2.set_xticks(log_lambdas[tick_indices])
ax2.set_xticklabels([str(n_nonzero[i]) for i in tick_indices])
ax2.set_xlabel('Number of Variables', fontsize=12)

# Ticks nach innen richten (wie in R)
# ax2.tick_params(axis='x', direction='in', pad=-15)
ax2.tick_params(axis='x', direction='out')#, pad=-15)
```

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

```
Performing detailed cross-validation for plotting…
```



1.2 Ridge Regression MSE

Prepare a slide with a table that reports training MSE and test MSE for different models. Fill in the MSE from the linear model using all features from Problem Set 1. Now compute the training and test MSE for the ridge regression with the optimal penalty parameter λ from *Q1.1*.

[14]:
```
# Fit Ridge regression with optimal lambda
ridge_optimal = Ridge(alpha=best_lambda_ridge)
ridge_optimal.fit(X_train, y_train)

# Calculate training and test MSE
y_pred_ridge_train = ridge_optimal.predict(X_train)
mse_ridge_train = mean_squared_error(y_train, y_pred_ridge_train)

y_pred_ridge_test = ridge_optimal.predict(X_test)
mse_ridge_test = mean_squared_error(y_test, y_pred_ridge_test)

print(f"Ridge Train MSE: {mse_ridge_train:.5f}")
print(f"Ridge Test MSE:  {mse_ridge_test:.5f}")
```

```
# Show coefficients
ridge_coefs = pd.DataFrame({
    'Variable': X_train.columns,
    'Ridge_Coefficient': ridge_optimal.coef_
})
print("\nRidge Coefficients:")
print(ridge_coefs)
```

```
Ridge Train MSE: 0.00511
Ridge Test MSE:  0.00878

Ridge Coefficients:
  Variable  Ridge_Coefficient
0       DP           0.086305
1       CS           0.064603
2     ntis          -0.260392
3      cay           0.389788
4       TS           0.327515
5     svar           0.156850
```

1.3 Lasso Regression

Redo the two tasks above using Lasso instead of Ridge. Again fix the seed to 2. Provide a plot of the cross-validation MSE as a function of $\log(\lambda)$ and interpret. Provide a table that shows the coefficient of the Lasso with the optimal penalty parameter λ. Compute the training and test MSE of this Lasso model and add it to the table from *Q1.2*.

[15]:
```
# Lasso with cross-validation
lasso_cv = LassoCV(alphas=lambda_grid, cv=5, random_state=2, max_iter=10000)
lasso_cv.fit(X_train, y_train)

# Get best lambda
best_lambda_lasso = lasso_cv.alpha_
print(f"Best lambda (Lasso): {best_lambda_lasso:.6f}")
print(f"Log of best lambda: {np.log(best_lambda_lasso):.4f}")
```

```
Best lambda (Lasso): 0.000152
Log of best lambda: -8.7917
```

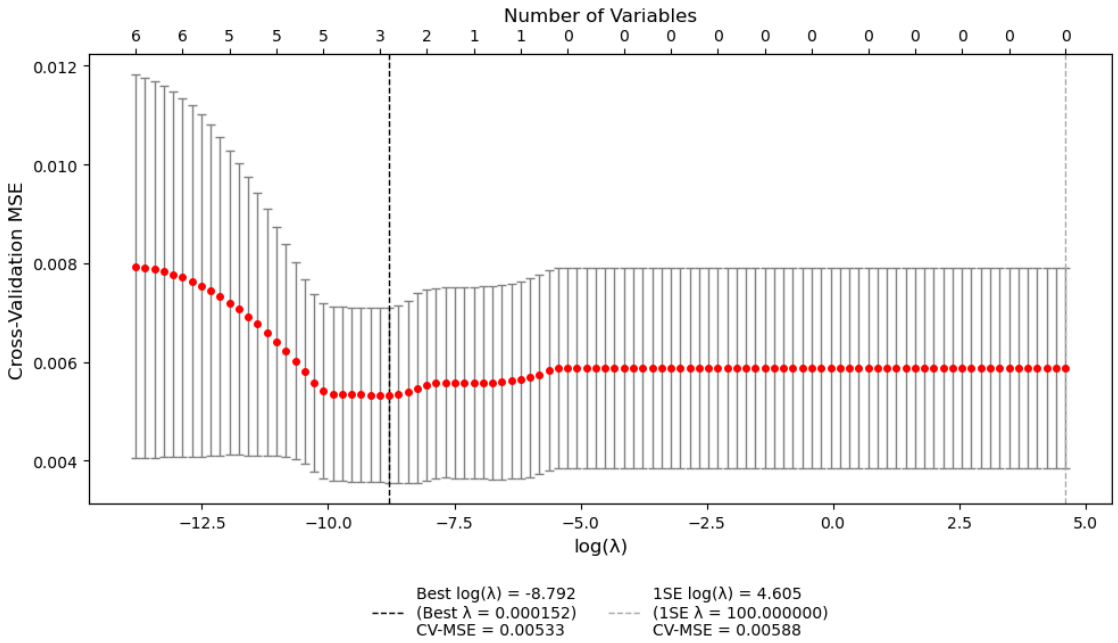
[16]:
```
# Manual cross-validation for plotting (to match R's detailed CV output)
print("Performing detailed cross-validation for Lasso plotting...")
cv_mse_lasso = []
cv_mse_lasso_std = []

for alpha in lambda_grid:
    lasso_temp = Lasso(alpha=alpha, random_state=2, max_iter=10000)
    cv_results = cross_validate(lasso_temp, X_train, y_train,
                                cv=5, scoring='neg_mean_squared_error',
```

```
                                return_train_score=False)
    cv_scores = -cv_results['test_score']  # Convert to positive MSE
    cv_mse_lasso.append(cv_scores.mean())
    cv_mse_lasso_std.append(cv_scores.std())

cv_mse_lasso = np.array(cv_mse_lasso)
cv_mse_lasso_std = np.array(cv_mse_lasso_std)

# Index von best_lambda_lasso im Grid finden:
idx_best_lasso = np.where(lambda_grid == best_lambda_lasso)[0][0]
best_mse_lasso = cv_mse_lasso[idx_best_lasso]

# Index für _1SE finden:
# Bestes MSE + 1SE-Grenze
threshold_lasso = best_mse_lasso + cv_mse_lasso_std[idx_best_lasso]
# Kandidaten-Lambdas finden, die <= threshold sind
candidates_lasso = np.where(cv_mse_lasso <= threshold_lasso)[0]
# Nimm das größte  (also den einfachsten / regularisiertesten Kandidaten)
idx_1se_lasso = candidates_lasso[-1]
best_lambda_1se_lasso = lambda_grid[idx_1se_lasso]
best_mse_1se_lasso = cv_mse_lasso[idx_1se_lasso]

# --- NEUER CODE FÜR y2-ACHSE: Anzahl Variablen berechnen ---
n_nonzero_lasso = []
for alpha in lambda_grid:
    lasso_temp = Lasso(alpha=alpha, random_state=2, max_iter=10000)
    lasso_temp.fit(X_train, y_train)
    # Für Lasso: Zähle nicht-null Koeffizienten
    n_nonzero_lasso.append(np.sum(np.abs(lasso_temp.coef_) > 1e-10))

plt.style.use('default')
# --- GEÄNDERT: fig, ax1 für subplot mit y2-Achse ---
fig, ax1 = plt.subplots(figsize=(10, 6))

ax1.errorbar(np.log(lambda_grid),
             cv_mse_lasso,
             yerr=cv_mse_lasso_std,
             capsize=3,
             color="red",          # Marker/Linie Rot
             ecolor="grey",        # Error Bars grau
             elinewidth=1,         # (optional) Breite der Error Bars
             fmt='o',              # (optional) schwarze '-'Linie + 'o'Punkte␣
 ↪'-o'
             markersize=4,
             )

ax1.axvline(np.log(best_lambda_lasso),
```

```
            color='black',
            linestyle='--',
            linewidth=1,
            label=(f'Best log( ) = {np.log(best_lambda_lasso):.3f}\n'
                   f'(Best   = {best_lambda_lasso:.6f})\n'
                   f'CV-MSE = {best_mse_lasso:.5f}'
                  )
            )

ax1.axvline(np.log(best_lambda_1se_lasso),
            color='darkgrey',
            linestyle='--',
            linewidth=1,
            label=(f'1SE log( ) = {np.log(best_lambda_1se_lasso):.3f}\n'
                   f'(1SE   = {best_lambda_1se_lasso:.6f})\n'
                   f'CV-MSE = {best_mse_1se_lasso:.5f}'
                  )
            )

ax1.set_xlabel('log( )', fontsize=12)
ax1.set_ylabel('Cross-Validation MSE', fontsize=12)
# ax1.set_title('Lasso Regression: Cross-Validation MSE vs log( )')
ax1.grid(False)

# --- Legende unter dem Plot, zentriert, nebeneinander ---
ax1.legend(bbox_to_anchor=(0.5, -0.15),
          loc='upper center',
          ncol=3,
          frameon=True,
          framealpha=0   # 0 = unsichtbar, 1 = voll sichtbar
          )

# --- NEUER CODE: y2-Achse oben für Anzahl Variablen ---
ax2 = ax1.twiny()
ax2.set_xlim(ax1.get_xlim())

# Ticks und Labels für die Anzahl der Variablen setzen
log_lambdas_lasso = np.log(lambda_grid)
# Zeige max 20 Ticks um Überlappung zu vermeiden
n_ticks = min(20, len(lambda_grid))
tick_indices = np.linspace(0, len(lambda_grid)-1, n_ticks, dtype=int)

ax2.set_xticks(log_lambdas_lasso[tick_indices])
ax2.set_xticklabels([str(n_nonzero_lasso[i]) for i in tick_indices])
ax2.set_xlabel('Number of Variables', fontsize=12)

# Ticks nach außen richten (wie in Ridge)
```

```
ax2.tick_params(axis='x', direction='out')

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

```
Performing detailed cross-validation for Lasso plotting…
```



[17]:
```
# Fit Lasso with optimal lambda
lasso_optimal = Lasso(alpha=best_lambda_lasso, random_state=2, max_iter=10000)
lasso_optimal.fit(X_train, y_train)

# Calculate training and test MSE
y_pred_lasso_train = lasso_optimal.predict(X_train)
mse_lasso_train = mean_squared_error(y_train, y_pred_lasso_train)

y_pred_lasso_test = lasso_optimal.predict(X_test)
mse_lasso_test = mean_squared_error(y_test, y_pred_lasso_test)

print(f"Lasso Train MSE: {mse_lasso_train:.5f}")
print(f"Lasso Test MSE:  {mse_lasso_test:.5f}")
```

```
Lasso Train MSE: 0.00524
Lasso Test MSE:  0.00916
```

[18]:
```
# Show coefficients
lasso_coefs = pd.DataFrame({
```

```
    'Variable': X_train.columns,
    'Lasso_Coefficient': lasso_optimal.coef_
})
print("Lasso Coefficients:")
print(lasso_coefs)

# Count non-zero coefficients
non_zero_coefs = np.sum(lasso_optimal.coef_ != 0)
print(f"\nNumber of non-zero coefficients: {non_zero_coefs}")
```

```
Lasso Coefficients:
  Variable  Lasso_Coefficient
0       DP           0.085394
1       CS           0.000000
2     ntis          -0.000000
3      cay           0.443235
4       TS           0.000000
5     svar           0.000000

Number of non-zero coefficients: 2
```

1.4 Sparse Lasso Regression (3 Variables)

Now suppose your boss tells you that he only trusts sparse models with few variables. Use the Lasso and choose the tuning parameter λ such that the model only considers 3 out of the six variables. Report the coefficients and compare them to the coefficients from the optimal model from *Q1.3* and interpret. Compute the training and test MSE of this Lasso model and add it to the table from *Q1.2*. Interpret.

[19]:
```
# Find lambda that gives exactly 3 non-zero coefficients
# We'll search through different lambda values
lambda_test_range = np.logspace(-4, -1, 1000)  # More focused range
n_features_list = []

for alpha in lambda_test_range:
    lasso_temp = Lasso(alpha=alpha, random_state=2, max_iter=10000)
    lasso_temp.fit(X_train, y_train)
    n_features = np.sum(lasso_temp.coef_ != 0)
    n_features_list.append(n_features)

# Find alpha that gives exactly 3 features
target_features = 3
suitable_alphas = [alpha for alpha, n_feat in zip(lambda_test_range,␣
 ↪n_features_list)
                   if n_feat == target_features]

if suitable_alphas:
    # Use the middle value from suitable alphas
```

```
    sparse_lambda = suitable_alphas[len(suitable_alphas)//2]
    print(f"Lambda for 3 variables: {sparse_lambda:.6f}")

    # Fit sparse Lasso
    lasso_sparse = Lasso(alpha=sparse_lambda, random_state=2, max_iter=10000)
    lasso_sparse.fit(X_train, y_train)

    # Calculate MSE
    y_pred_lasso_sparse_train = lasso_sparse.predict(X_train)
    mse_lasso_sparse_train = mean_squared_error(y_train,␣
 ↪y_pred_lasso_sparse_train)

    y_pred_lasso_sparse_test = lasso_sparse.predict(X_test)
    mse_lasso_sparse_test = mean_squared_error(y_test, y_pred_lasso_sparse_test)

    print(f"Sparse Lasso Train MSE: {mse_lasso_sparse_train:.5f}")
    print(f"Sparse Lasso Test MSE:  {mse_lasso_sparse_test:.5f}")

    # Show coefficients
    sparse_lasso_coefs = pd.DataFrame({
        'Variable': X_train.columns,
        'Sparse_Lasso_Coefficient': lasso_sparse.coef_
    })
    print("\nSparse Lasso Coefficients (3 variables):")
    print(sparse_lasso_coefs)

    # Show which variables are selected
    selected_vars = X_train.columns[lasso_sparse.coef_ != 0].tolist()
    print(f"\nSelected variables: {selected_vars}")

else:
    print("Could not find lambda that gives exactly 3 variables")
    # Use a reasonable approximation
    sparse_lambda = 0.0125  # From R code
    lasso_sparse = Lasso(alpha=sparse_lambda, random_state=2, max_iter=10000)
    lasso_sparse.fit(X_train, y_train)

    y_pred_lasso_sparse_train = lasso_sparse.predict(X_train)
    mse_lasso_sparse_train = mean_squared_error(y_train,␣
 ↪y_pred_lasso_sparse_train)

    y_pred_lasso_sparse_test = lasso_sparse.predict(X_test)
    mse_lasso_sparse_test = mean_squared_error(y_test, y_pred_lasso_sparse_test)

    print(f"Sparse Lasso Train MSE: {mse_lasso_sparse_train:.5f}")
    print(f"Sparse Lasso Test MSE:  {mse_lasso_sparse_test:.5f}")
```

```
    n_selected = np.sum(lasso_sparse.coef_ != 0)
    print(f"Number of selected variables: {n_selected}")
```

```
Lambda for 3 variables: 0.000128
Sparse Lasso Train MSE: 0.00522
Sparse Lasso Test MSE:  0.00925

Sparse Lasso Coefficients (3 variables):
  Variable  Sparse_Lasso_Coefficient
0       DP                 0.086918
1       CS                 0.000000
2     ntis                -0.000000
3      cay                 0.481694
4       TS                 0.023727
5     svar                 0.000000

Selected variables: ['DP', 'cay', 'TS']
```

[20]:
```
# Comprehensive results table
results_dict = {
    'Model': ['Linear Regression', 'Ridge Regression', 'Lasso Regression',␣
 ↪'Sparse Lasso (3 vars)'],
    'In-sample MSE': [mse_train_lm, mse_ridge_train, mse_lasso_train,␣
 ↪mse_lasso_sparse_train],
    'Out-of-sample MSE': [mse_test_lm, mse_ridge_test, mse_lasso_test,␣
 ↪mse_lasso_sparse_test]
}

results_df = pd.DataFrame(results_dict)
results_df = results_df.round(5)

print(results_df.to_string(index=False))
```

```
               Model  In-sample MSE  Out-of-sample MSE
   Linear Regression        0.00505            0.00861
    Ridge Regression        0.00511            0.00878
    Lasso Regression        0.00524            0.00916
Sparse Lasso (3 vars)        0.00522            0.00925
```

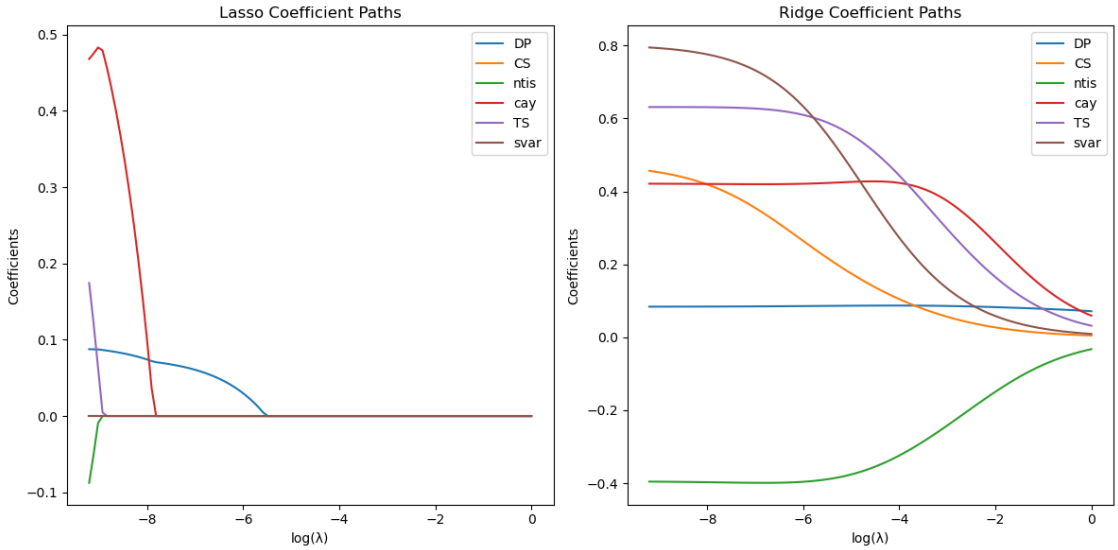
[21]:
```
# Compare table of all coefficients
coef_comparison = pd.DataFrame({
    'Variable': X_train.columns,
    'Linear': model_all.coef_,
    'Ridge': ridge_optimal.coef_,
    'Lasso_Optimal': lasso_optimal.coef_,
    'Lasso_Sparse': lasso_sparse.coef_
}).round(6)
```

```
print(coef_comparison.to_string(index=False))
```

```
Variable    Linear     Ridge  Lasso_Optimal  Lasso_Sparse
      DP  0.083873  0.086305       0.085394      0.086918
      CS  0.474950  0.064603       0.000000      0.000000
    ntis -0.394479 -0.260392      -0.000000     -0.000000
     cay  0.421456  0.389788       0.443235      0.481694
      TS  0.631040  0.327515       0.000000      0.023727
    svar  0.802650  0.156850       0.000000      0.000000
```

[26]:
```
# VISUALIZATION: COEFFICIENT PATHS  -  (EXTRA)
# Lasso coefficient path
lasso_path = Lasso(random_state=2, max_iter=10000)
lambda_path = np.logspace(-4, 0, 100)
coefs_lasso = []

for alpha in lambda_path:
    lasso_path.set_params(alpha=alpha)
    lasso_path.fit(X_train, y_train)
    coefs_lasso.append(lasso_path.coef_.copy())

coefs_lasso = np.array(coefs_lasso)

plt.figure(figsize=(12, 6))

# Lasso path
plt.subplot(1, 2, 1)
for i, feature in enumerate(X_train.columns):
    plt.plot(np.log(lambda_path), coefs_lasso[:, i], label=feature)
plt.xlabel('log( )')
plt.ylabel('Coefficients')
plt.title('Lasso Coefficient Paths')
plt.legend()
plt.grid(False)

# Ridge path
ridge_path = Ridge()
coefs_ridge = []

for alpha in lambda_path:
    ridge_path.set_params(alpha=alpha)
    ridge_path.fit(X_train, y_train)
    coefs_ridge.append(ridge_path.coef_.copy())

coefs_ridge = np.array(coefs_ridge)

plt.subplot(1, 2, 2)
```

```
for i, feature in enumerate(X_train.columns):
    plt.plot(np.log(lambda_path), coefs_ridge[:, i], label=feature)
plt.xlabel('log( )')
plt.ylabel('Coefficients')
plt.title('Ridge Coefficient Paths')
plt.legend()
plt.grid(False)

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



Question 2: Tree-Based Methods

2.1 Large Regression Tree

Fit a large regression tree using the training data. Report the number of terminal nodes as well as the most important variables for splitting the tree.

[27]:
```
# Loading needed packages for Question 2 tasks
from sklearn.tree import DecisionTreeRegressor, plot_tree, export_text
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score, validation_curve
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error
import seaborn as sns
```

[28]:
```
# Fit a large regression tree (minimal restrictions to allow deep tree)
# In sklearn, we need to set parameters to allow a "large" tree similar to R's␣
 ↪tree()
large_tree = DecisionTreeRegressor(
    random_state=2,
    min_samples_split=2,      # Minimum samples to split (very low to allow␣
 ↪deep tree)
    min_samples_leaf=1,       # Minimum samples in leaf (very low)
    max_depth=None,           # No depth limit initially
    min_impurity_decrease=0   # No minimum impurity decrease required
)

large_tree.fit(X_train, y_train)

# Get tree information
n_nodes = large_tree.tree_.node_count
n_leaves = large_tree.get_n_leaves()
max_depth = large_tree.get_depth()

print(f"Number of nodes: {n_nodes}")
print(f"Number of terminal nodes (leaves): {n_leaves}")
print(f"Maximum depth: {max_depth}")
```

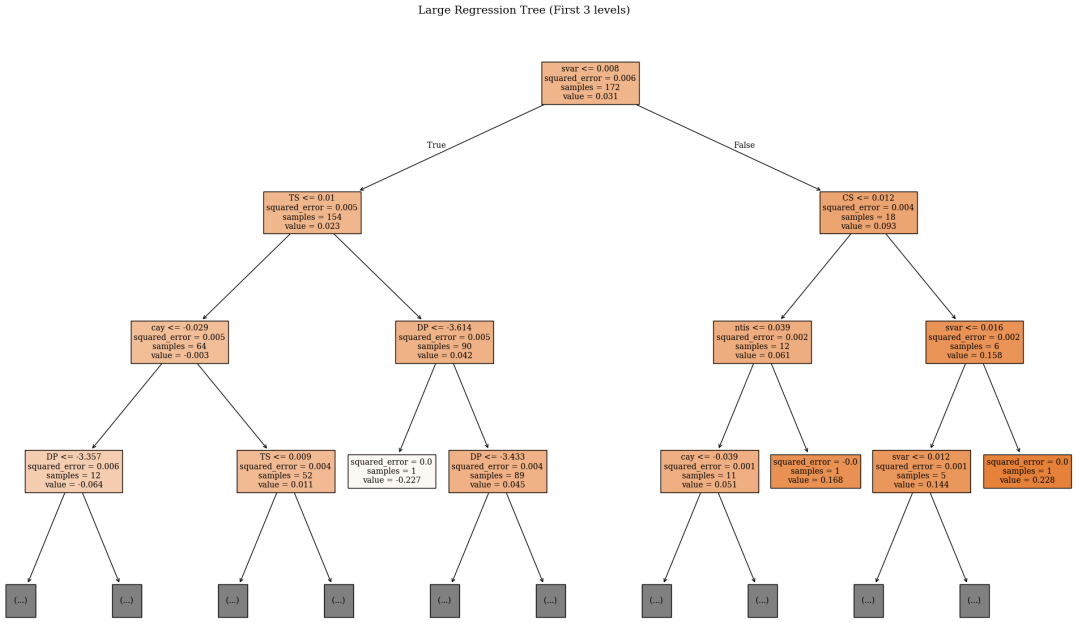
```
Number of nodes: 343
Number of terminal nodes (leaves): 172
Maximum depth: 16
```

[29]:
```
# Feature importance (equivalent to R's splitting importance)
feature_importance = pd.DataFrame({
    'Variable': X_train.columns,
    'Importance': large_tree.feature_importances_
}).sort_values('Importance', ascending=False)

print("Feature Importance (most important variables for splitting):")
print(feature_importance)

# Most important variable
most_important_var = feature_importance.iloc[0]['Variable']
print(f"\nMost important variable for splitting: {most_important_var}")
```

```
Feature Importance (most important variables for splitting):
  Variable  Importance
4       TS    0.233574
0       DP    0.228339
5     svar    0.194723
1       CS    0.186572
3      cay    0.117219
2     ntis    0.039573
```

```
Most important variable for splitting: TS
```

[21]:
```
# Visualize the tree (showing only top levels due to size)
plt.figure(figsize=(20, 12))
plot_tree(large_tree,
          feature_names=X_train.columns,
          filled=True,
          max_depth=3,  # Show only first 3 levels for readability
          fontsize=10)
plt.title("Large Regression Tree (First 3 levels)")
plt.tight_layout()
plt.show()
```



2.2 Train and Test MSE for the Large Tree

Compute the training and test MSE of the tree and add it to the table from *Q1.2*.

[30]:
```
# Calculate training MSE
y_pred_large_tree_train = large_tree.predict(X_train)
mse_large_tree_train = mean_squared_error(y_train, y_pred_large_tree_train)

# Calculate test MSE
y_pred_large_tree_test = large_tree.predict(X_test)
mse_large_tree_test = mean_squared_error(y_test, y_pred_large_tree_test)
```

```
print(f"Large Tree Train MSE: {mse_large_tree_train:.6f}")
print(f"Large Tree Test MSE:  {mse_large_tree_test:.6f}")
```

```
Large Tree Train MSE: 0.000000
Large Tree Test MSE:  0.045659
```

2.3. Cross-Validation for optimal Tree Pruning

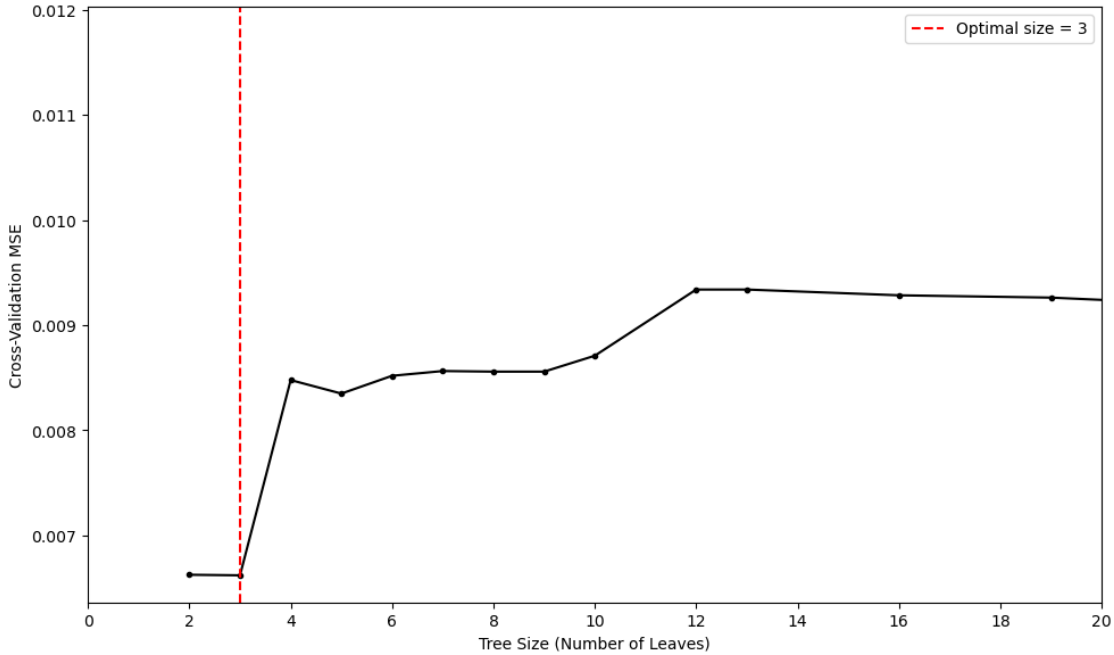
Again set the seed to 2 and use 5-fold cross validation to determine the optimal pruning parameter for the large tree. Provide a plot of the prediction error against the size of the tree. Report the optimal tree size and provide a plot of the pruned tree. Which variables are important for splitting the pruned tree?

[31]:
```
# Set seed for reproducibility
np.random.seed(2)

# In sklearn, pruning is done by varying max_leaf_nodes or ccp_alpha
# We'll use cost complexity pruning (ccp_alpha) which is more similar to R's␣
 ↪approach

# First, get the cost complexity pruning path
path = large_tree.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

# Remove the last alpha value (it would result in a tree with only root)
ccp_alphas = ccp_alphas[:-1]

# Perform cross-validation for different alpha values
cv_scores = []
tree_sizes = []

for ccp_alpha in ccp_alphas:
    tree_temp = DecisionTreeRegressor(random_state=2, ccp_alpha=ccp_alpha)
    tree_temp.fit(X_train, y_train)

    # Get tree size (number of leaves)
    tree_sizes.append(tree_temp.get_n_leaves())

    # 5-fold cross-validation
    scores = cross_val_score(tree_temp, X_train, y_train,
                             cv=5, scoring='neg_mean_squared_error')
    cv_scores.append(-scores.mean())  # Convert back to positive MSE

cv_scores = np.array(cv_scores)
tree_sizes = np.array(tree_sizes)

# Find the optimal tree size (minimum CV error)
best_idx = np.argmin(cv_scores)
```

```
best_size = tree_sizes[best_idx]
best_alpha = ccp_alphas[best_idx]

print(f"Optimal tree size (number of leaves): {best_size}")
print(f"Optimal ccp_alpha: {best_alpha:.6f}")
```

```
Optimal tree size (number of leaves): 3
Optimal ccp_alpha: 0.000424
```

[32]:
```
from matplotlib.ticker import MultipleLocator

# Plot prediction error against tree size
plt.style.use('default')
plt.figure(figsize=(10, 6))
plt.plot(tree_sizes, cv_scores, '-o', color='black', markersize=3)
plt.xlim(0, 20)
# X-Ticks in 2er Schritten (nur ganze Zahlen)
plt.gca().xaxis.set_major_locator(MultipleLocator(2))
plt.axvline(x=best_size, color='red', linestyle='--',
            label=f'Optimal size = {best_size}')
plt.xlabel('Tree Size (Number of Leaves)')
plt.ylabel('Cross-Validation MSE')
# plt.title('Prediction Error vs Tree Size')
plt.legend()
plt.grid(False)
plt.tight_layout()
plt.show()
```



[33]:
```
# Fit the pruned tree with optimal alpha
pruned_tree = DecisionTreeRegressor(random_state=2, ccp_alpha=best_alpha)
pruned_tree.fit(X_train, y_train)

print(f"Pruned tree statistics:")
print(f"  Number of leaves: {pruned_tree.get_n_leaves()}")
print(f"  Maximum depth: {pruned_tree.get_depth()}")

# Feature importance for pruned tree
pruned_importance = pd.DataFrame({
    'Variable': X_train.columns,
    'Importance': pruned_tree.feature_importances_
}).sort_values('Importance', ascending=False)
```

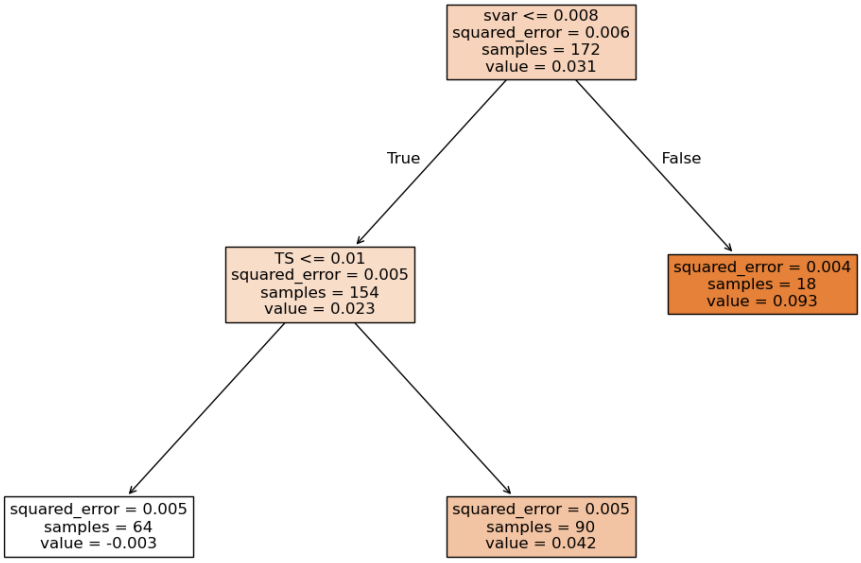
```
Pruned tree statistics:
  Number of leaves: 3
  Maximum depth: 2
```

[34]:
```
print("Feature Importance for Pruned Tree:")
print(pruned_importance)

# Variables used in splitting (non-zero importance)
important_vars = pruned_importance[pruned_importance['Importance'] >␣
 ↪0]['Variable'].tolist()
print(f"\nVariables important for splitting the pruned tree: {important_vars}")
```

```
Feature Importance for Pruned Tree:
  Variable  Importance
5     svar    0.511469
4       TS    0.488531
1       CS    0.000000
0       DP    0.000000
3      cay    0.000000
2     ntis    0.000000

Variables important for splitting the pruned tree: ['svar', 'TS']
```

[35]:
```
# Visualize the pruned tree
plt.style.use('default')
plt.figure(figsize=(12, 8))
plot_tree(pruned_tree,
          feature_names=X_train.columns,
          filled=True,
          fontsize=12)
# plt.title("Pruned Regression Tree")
plt.tight_layout()
```

```
plt.show()
```



2.4. Train and Test MSE for Pruned Tree

Compute the training and test MSE of the pruned tree and add it to the table from *Q1.2*.

[36]:
```
# Calculate training MSE for pruned tree
y_pred_pruned_tree_train = pruned_tree.predict(X_train)
mse_pruned_tree_train = mean_squared_error(y_train, y_pred_pruned_tree_train)

# Calculate test MSE for pruned tree
y_pred_pruned_tree_test = pruned_tree.predict(X_test)
mse_pruned_tree_test = mean_squared_error(y_test, y_pred_pruned_tree_test)

print(f"Pruned Tree Train MSE: {mse_pruned_tree_train:.6f}")
print(f"Pruned Tree Test MSE:  {mse_pruned_tree_test:.6f}")
```

```
Pruned Tree Train MSE: 0.004898
Pruned Tree Test MSE:  0.008131
```

2.5. Random Forest

Finally, use random forest to improve the predictions. Motivate your choice for the tuning parameters. Report the training and test MSE and add it to the table from *Q1.2*. Which variables are

most important in the random forest?

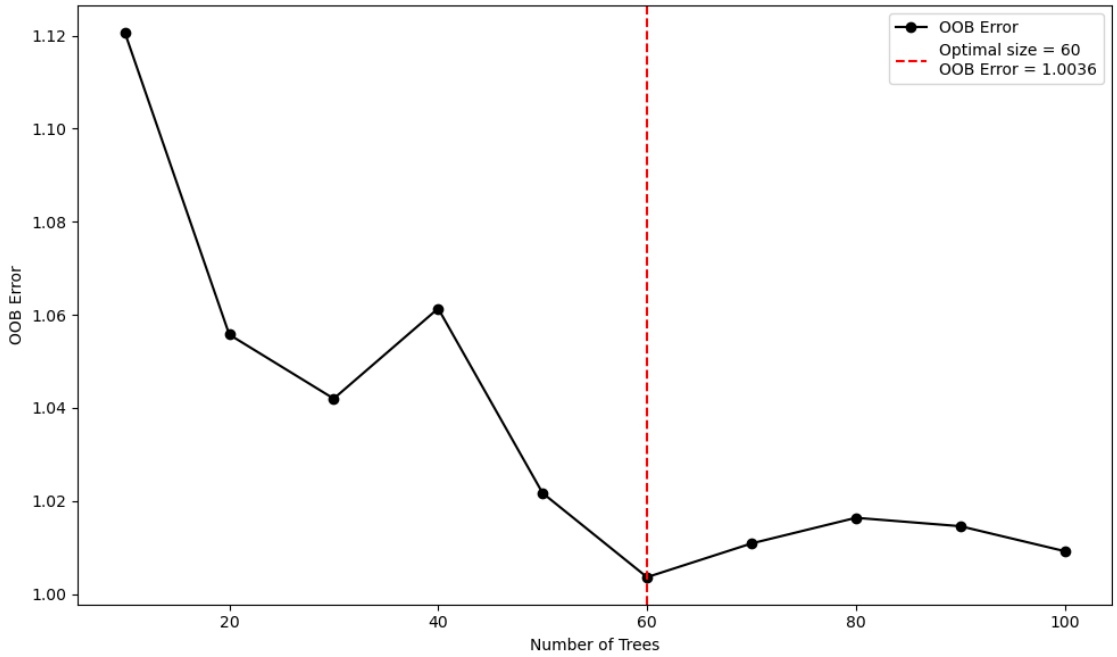
[37]:
```
# Random Forest with tuned parameters
rf_model = RandomForestRegressor(
    n_estimators=100,        # corresponds tontree in R
    max_features=2,          # corresponds tomtry in R (number of features to␣
 ↪consider at each split)
    random_state=2,
    n_jobs=-1,               # Use all available cores
    oob_score=True
)

rf_model.fit(X_train, y_train)

print(f"Random Forest Parameters:")
print(f"  Number of trees (n_estimators): {rf_model.n_estimators}")
print(f"  Features considered per split (max_features): {rf_model.
 ↪max_features}")
print(f"  Out-of-bag score: {rf_model.oob_score_}")

# Parameter justification
print(f"\nParameter Justification:")
print(f"  - n_estimators=100: Sufficient trees for stable predictions without␣
 ↪overfitting")
print(f"  - max_features=2: For regression, typically sqrt(p) or p/3, where␣
 ↪p={len(X_train.columns)}")
print(f"    sqrt({len(X_train.columns)}) = {int(np.sqrt(len(X_train.
 ↪columns)))}, so 2 is reasonable")
```

```
Random Forest Parameters:
  Number of trees (n_estimators): 100
  Features considered per split (max_features): 2
  Out-of-bag score: -0.009214729335368155

Parameter Justification:
  - n_estimators=100: Sufficient trees for stable predictions without
overfitting
  - max_features=2: For regression, typically sqrt(p) or p/3, where p=6
    sqrt(6) = 2, so 2 is reasonable
```

[38]:
```
# Plot OOB error evolution (if OOB scoring is enabled)
# Note: In "RandomForestRegressor", "oob_score_" represents OOB-R^2␣
 ↪(Bestimmtheitsmaß), not the OOB-MSE.
# Note: sklearn doesn't store OOB evolution by default, so we'll create a␣
 ↪simple version
oob_errors = []
for n_est in range(10, 101, 10):
```

```
    rf_temp = RandomForestRegressor(n_estimators=n_est, max_features=2,
                                    random_state=2, oob_score=True)
    rf_temp.fit(X_train, y_train)
    oob_errors.append(1 - rf_temp.oob_score_)  # Convert R^2 to "error": (1 - R^2)

best_idx = np.argmin(oob_errors)
best_size = list(range(10, 101, 10))[best_idx]
best_oob_error = oob_errors[best_idx]

plt.style.use('default')
plt.figure(figsize=(10, 6))
plt.plot(range(10, 101, 10), oob_errors, '-o', color='black', label="OOB Error")
plt.axvline(x=best_size, color='red', linestyle='--',
            label=f'Optimal size = {best_size}\nOOB Error = {best_oob_error:.4f}')

plt.xlabel('Number of Trees')
plt.ylabel('OOB Error')
# plt.title('Random Forest: OOB Error vs Number of Trees')
plt.grid(False)
plt.legend()
plt.tight_layout()
plt.show()
```

```
/usr/lib/python3.13/site-packages/sklearn/ensemble/_forest.py:611: UserWarning:
Some inputs do not have OOB scores. This probably means too few trees were used
to compute any reliable OOB estimates.
  warn(
```



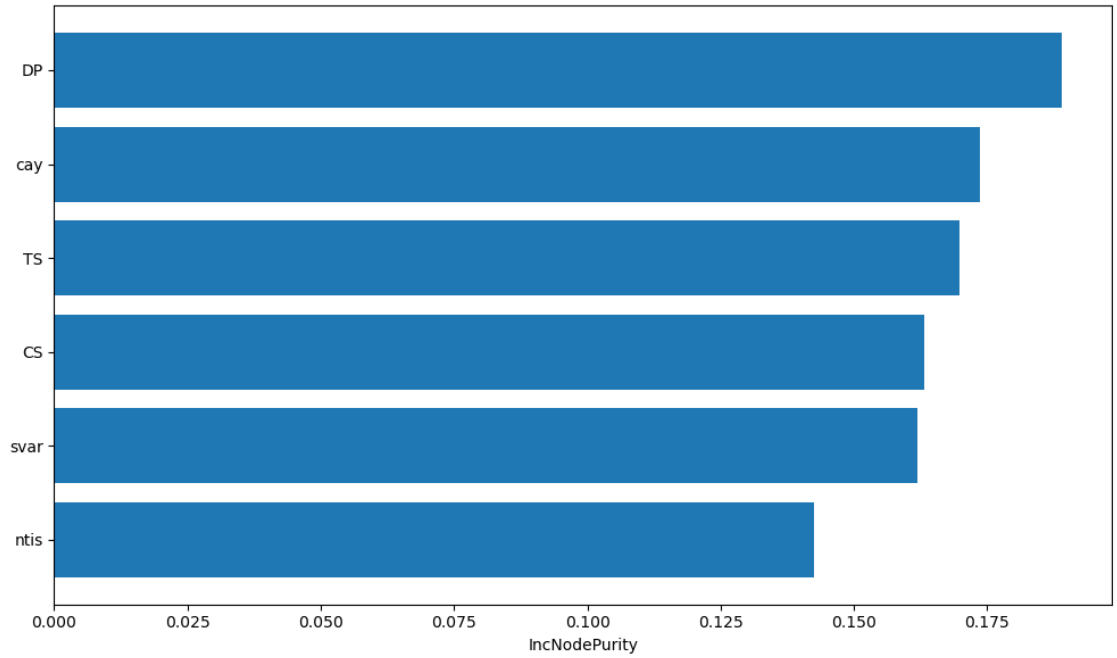
[39]:
```python
# Calculate training and test MSE
y_pred_rf_train = rf_model.predict(X_train)
mse_rf_train = mean_squared_error(y_train, y_pred_rf_train)

y_pred_rf_test = rf_model.predict(X_test)
mse_rf_test = mean_squared_error(y_test, y_pred_rf_test)

print(f"\nRandom Forest Train MSE: {mse_rf_train:.6f}")
print(f"Random Forest Test MSE: {mse_rf_test:.6f}")
```

```
Random Forest Train MSE: 0.000739
Random Forest Test MSE: 0.012266
```

[40]:
```python
# Feature importance
rf_importance = pd.DataFrame({
    'Variable': X_train.columns,
    'Importance': rf_model.feature_importances_
}).sort_values('Importance', ascending=False)

print("Feature Importance (Random Forest):")
print(rf_importance)

# Most important variable
most_important_rf = rf_importance.iloc[0]['Variable']
print(f"\nMost important variable in Random Forest: {most_important_rf}")
```

```
Feature Importance (Random Forest):
  Variable  Importance
0       DP    0.188921
3      cay    0.173580
4       TS    0.169850
1       CS    0.163270
5     svar    0.161956
2     ntis    0.142423

Most important variable in Random Forest: DP
```

[41]:
```python
# Variable importance plot
plt.style.use('default')
plt.figure(figsize=(10, 6))
plt.barh(range(len(rf_importance)), rf_importance['Importance'])
plt.yticks(range(len(rf_importance)), rf_importance['Variable'])
plt.xlabel('IncNodePurity')
# plt.title('Random Forest: Variable Importance Plot')
plt.gca().invert_yaxis()  # Highest importance at top
plt.tight_layout()
plt.show()
```



The test set MSE associated with the bagged regression tree is 0.0186…

[43]:
```python
# COMPREHENSIVE RESULTS TABLE
results_dict = {
```

```
    'Model': [
        'Linear Regression',
        'Ridge Regression',
        'Lasso Regression',
        'Sparse Lasso (3 vars)',
        'Large Regression Tree',
        'Pruned Regression Tree',
        'Random Forest'
    ],
    'In-sample MSE': [
        mse_train_lm,
        mse_ridge_train,
        mse_lasso_train,
        mse_lasso_sparse_train,
        mse_large_tree_train,
        mse_pruned_tree_train,
        mse_rf_train
    ],
    'Out-of-sample MSE': [
        mse_test_lm,
        mse_ridge_test,
        mse_lasso_test,
        mse_lasso_sparse_test,
        mse_large_tree_test,
        mse_pruned_tree_test,
        mse_rf_test
    ]
}

comprehensive_results = pd.DataFrame(results_dict).round(6)

print(comprehensive_results.to_string(index=False))
```

```
                Model  In-sample MSE  Out-of-sample MSE
    Linear Regression       0.005047           0.008608
     Ridge Regression       0.005111           0.008779
     Lasso Regression       0.005235           0.009159
 Sparse Lasso (3 vars)       0.005218           0.009250
 Large Regression Tree       0.000000           0.045659
Pruned Regression Tree       0.004898           0.008131
         Random Forest       0.000739           0.012266
```

2.6. Model selection analysis

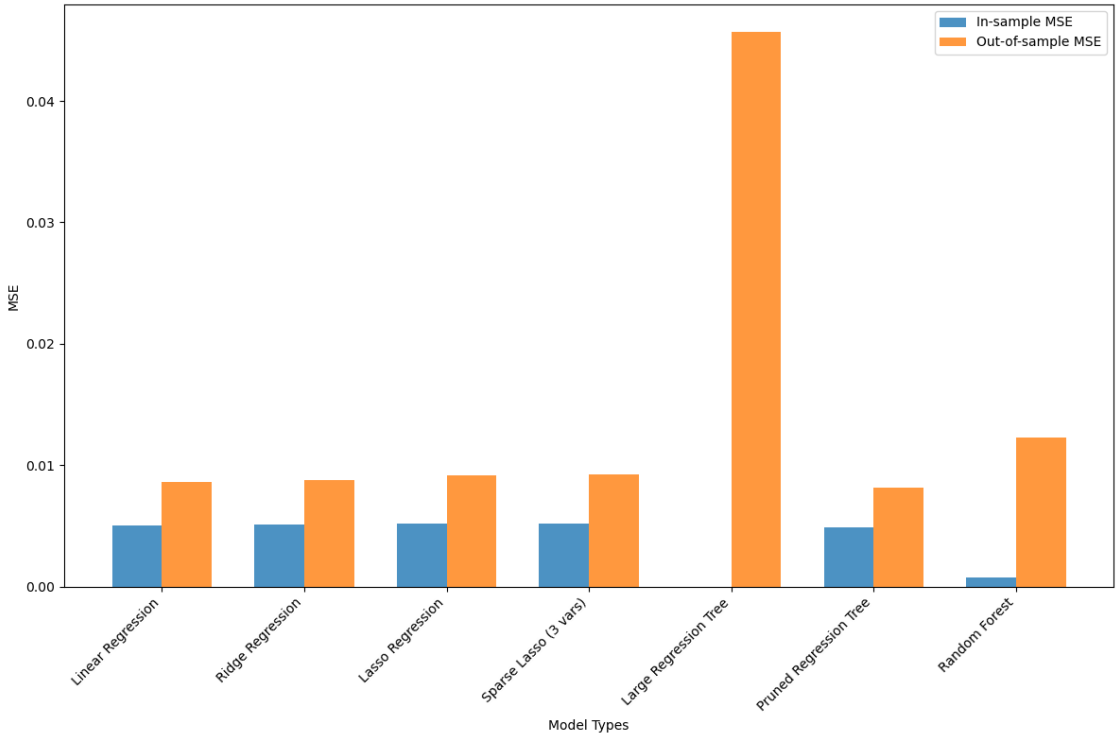
Supposed it is the beginning of 2020 and you have access to both the in-sample and out-of-sample errors for the different methods. Which model do you choose to predict stock markets in the future and why?

[45]:
```
# Analysis of overfitting vs generalization
comprehensive_results['Overfitting_Gap'] =␣
 ↪(comprehensive_results['Out-of-sample MSE'] -
                                          comprehensive_results['In-sample␣
 ↪MSE'])

print("Overfitting Analysis (Out-of-sample MSE - In-sample MSE):")
print(comprehensive_results[['Model', 'Overfitting_Gap']].
 ↪sort_values('Overfitting_Gap'))

# Find best model based on out-of-sample performance
best_model_idx = comprehensive_results['Out-of-sample MSE'].idxmin()
best_model = comprehensive_results.loc[best_model_idx, 'Model']
best_oos_mse = comprehensive_results.loc[best_model_idx, 'Out-of-sample MSE']

print(f"\nRecommended Model for Future Predictions:")
print(f"  Model: {best_model}")
print(f"  Out-of-sample MSE: {best_oos_mse:.6f}")

print(f"\nJustification:")
if best_model == 'Sparse Lasso (3 vars)':
    print("  - Lowest out-of-sample MSE indicates best generalization")
    print("  - Sparse model is interpretable and less prone to overfitting")
    print("  - Good balance between bias and variance")
elif best_model == 'Pruned Regression Tree':
    print("  - Good balance between complexity and generalization")
    print("  - Pruning reduces overfitting compared to large tree")
    print("  - Interpretable model structure")
else:
    print("  - Best out-of-sample performance")
    print("  - Shows good generalization to unseen data")
```

```
Overfitting Analysis (Out-of-sample MSE - In-sample MSE):
                    Model  Overfitting_Gap
5  Pruned Regression Tree         0.003233
0       Linear Regression         0.003561
1        Ridge Regression         0.003668
2        Lasso Regression         0.003924
3   Sparse Lasso (3 vars)         0.004032
6           Random Forest         0.011527
4   Large Regression Tree         0.045659

Recommended Model for Future Predictions:
  Model: Pruned Regression Tree
  Out-of-sample MSE: 0.008131

Justification:
```

- Good balance between complexity and generalization
- Pruning reduces overfitting compared to large tree
- Interpretable model structure

[46]:
```
# Visualization of model performance comparison
plt.style.use('default')
plt.figure(figsize=(12, 8))

models = comprehensive_results['Model']
train_mse = comprehensive_results['In-sample MSE']
test_mse = comprehensive_results['Out-of-sample MSE']

x = np.arange(len(models))
width = 0.35

plt.bar(x - width/2, train_mse, width, label='In-sample MSE', alpha=0.8)
plt.bar(x + width/2, test_mse, width, label='Out-of-sample MSE', alpha=0.8)

plt.xlabel('Model Types')
plt.ylabel('MSE')
# plt.title('Model Performance Comparison: In-sample vs Out-of-sample MSE')
plt.xticks(x, models, rotation=45, ha='right')
plt.legend()
plt.grid(False)
plt.tight_layout()
plt.show()
```



Appendix

The dataset contains the following variables:

- **ret**: the quarterly return of the US stock market (a number of 0.01 is a 1% return per quarter)
- **date**: the date in format *yyyyq* (19941 means the first quarter of 1994)
- **DP**: the dividend to price ratio of the stock market (a valuation measure whether prices are high or low relative to the dividends payed)
- **CS**: the credit spread defined as the difference in yields between high rated corporate bonds (save investments) and low rated corporate bonds (corporations that might go bankrupt). CS measures the additional return investors require to invest in risky firms compared to well established firms with lower risks
- **ntis**: A measure for corporate issuing activity (IPO's, stock repurchases,...)
- **cay**: a measure of the wealth-to-consumption ratio (how much is consumed relative to total wealth)
- **TS**: the term spread is the difference between the long term yield on government bonds and short term yields.
- **svar**: a measure for the stock market variance

For a full description of the data, see *Welch und Goyal* (2007). Google is also very helpful if you are interested in obtaining more intuition about the variables.

References

Welch, I. and A. Goyal (2007, 03). A Comprehensive Look at The Empirical Performance of Equity Premium Prediction. *The Review of Financial Studies 21* (4), 1455 – 1508.