Machine Learning for Economics and Finance

Problem Set 1 - Solution

Ole Wilms

July 29, 2024

Important Instructions

- The purpose of this tutorial is for you to practise some of the key concepts we covered in the first weeks
- In case you struggle with some problems, please post your questions on the Canvas discussion board
- For this exercise, NO write-up of your answers or submission is required. However, I recommend you already begin developing clean programs that you can use later in the take-home exam
- We will discuss the solutions for the problem set in the lecture on `DAY MONTH`

Setup

The main task of this problem set is to forecast the return of the US stock market. For this we will use the dataset `stockmarketdata.RDS` taken from Welch and Goyal (2007) which is available on *OpenOlat*. The dataset contains quarterly returns of the US stock market (ret) as well as several other variables that have been proposed by finance researchers to predict stock returns. A list of all variables together with a description can be found in the appendix. For example for quarter $1999Q1$ (date = 19991) it contains variables like the return of the stock market (ret_t), the dividend-to-price ratio (DP_t), the credit spread (CS_t) and so on. As the goal is to predict returns in the subsequent quarter, we are interested in models of the form

$$ret_{t+1} = f(DP_t, CS_t, ...) + \epsilon_{t+1}$$

Suppose you are an asset manager and it is the end of 1994, that is, you have all the data before 1995 available to train and validate your model. Your goal is to build a model that not only works in-sample, but can also predict returns in the future (after 1995).

Preliminaries

- Laden notwendiger Pakete und Einlesen der `stockmarketdata.rds` Datei

[6]:
```
# Loading the necessary packages for this exercise
import pyreadr                                                    # Package
 ↪for reading RDS files - https://github.com/ofajardo/pyreadr
import pandas as pd                                               # Package
 ↪for processing, analyzing, and visualizing data
import numpy as np                                                # Package
 ↪for handling vectors, matrices, or generally large multidimensional arrays
import statsmodels.api as sm                                      # Package
 ↪for investigating/estimating
from statsmodels.formula.api import ols
from sklearn.linear_model import LinearRegression, LogisticRegression # Package
 ↪for various classifications-, regressions- and Clustering-Algorithms
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import r2_score, mean_squared_error, accuracy_score


import matplotlib.pyplot as plt                                   # Package
 ↪for creating data visualizations
import seaborn as sns                                             #
 ↪Supplementary package to "matplotlib". (Modernizes design & simplere syntax)

# Defaults for the following matplotlib figures:
plt.rcParams.update({
    'figure.figsize': (10, 8),
    'font.family': 'serif',
    'font.size': 11,
    'axes.titlesize': 14,
    'axes.grid': False,
    'lines.linewidth': 2
})
```

[7]:
```
# Setup of the data set
df = pyreadr.read_r('stockmarketdata.rds')
df = df[None]  # Extrahieren des verfügbaren pandas DataFrame Objekts.

df.head()  # Showing the first five rows of the DataFrame.
```

[7]:

	date	ret	DP	CS	ntis	cay	TS	svar
0	19291.0	0.050490	-3.367688	0.010357	0.079805	NaN	-0.0083	0.007982
1	19292.0	0.087235	-3.412851	0.011105	0.116197	NaN	-0.0113	0.008405
2	19293.0	0.091067	-3.468392	0.012517	0.121390	NaN	-0.0083	0.008056
3	19294.0	-0.268418	-3.096184	0.012155	0.163522	NaN	0.0037	0.100171
4	19301.0	0.165884	-3.252345	0.010554	0.145496	NaN	0.0040	0.004662

[8]:
```
# Definition of a custom function for reading and reformatting the "date" year-quarter time data.
def convert_to_quarterly_date(numeric_date):
    """
    Converts a numeric date representing year and quarter into a quarterly date string in the format 'YYYY-Q'.

    Parameters:
    numeric_date (int or float): Numeric date representing year and quarter.
    The whole part indicates the year, and the decimal part indicates the quarter (e.g., 20191.0 for the first quarter of 2019).

    Returns:
    str: A string representing the quarterly date in the format 'YYYY-Q', where 'YYYY' is the year and 'Q' is the quarter.

    Example:
    >>> convert_to_quarterly_date(20191.0)
    '2019-Q1'
    """
    year = int(numeric_date) // 10        # Extracting the year information
    quarter = int(numeric_date) % 10      # Extracting the quarter information using the modulo operation
    quarter_str = f'Q{quarter}'           # Converting the integer quarter data to string format
    return f'{year}-Q{quarter}'           # Returning the desired string


# Applying the function to the "date" variable.
df['date'] = df['date'].apply(convert_to_quarterly_date)

df.tail()
```

[8]:

	date	ret	DP	CS	ntis	cay	TS	svar
360	2019-Q1	0.137489	-3.943400	0.010258	-0.023230	-0.039336	0.0017	0.004651
361	2019-Q2	0.042688	-3.960033	0.010006	-0.012562	-0.033844	-0.0010	0.003271
362	2019-Q3	0.017042	-3.951689	0.008505	-0.010862	-0.029529	-0.0019	0.005517
363	2019-Q4	0.090143	-4.015896	0.008410	-0.007222	-0.033609	0.0032	0.002319
364	2020-Q1	-0.193794	-3.769992	0.012252	-0.007731	-0.050141	0.0058	0.079049

Exploration und Visualisierung des Datensatzes (extra)

[9]:
```
df.info()  # The last column "Dtype" is of particular interest here.
           # The variable "date" is of type "object", therefore not a numerical␣
 ↪variable.
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 365 entries, 0 to 364
Data columns (total 8 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   date    365 non-null    object
 1   ret     365 non-null    float64
 2   DP      365 non-null    float64
 3   CS      365 non-null    float64
 4   ntis    365 non-null    float64
 5   cay     273 non-null    float64
 6   TS      365 non-null    float64
 7   svar    365 non-null    float64
dtypes: float64(7), object(1)
memory usage: 22.9+ KB
```

[10]:
```
df.describe().T   # Descriptive Statistics of the Data
```

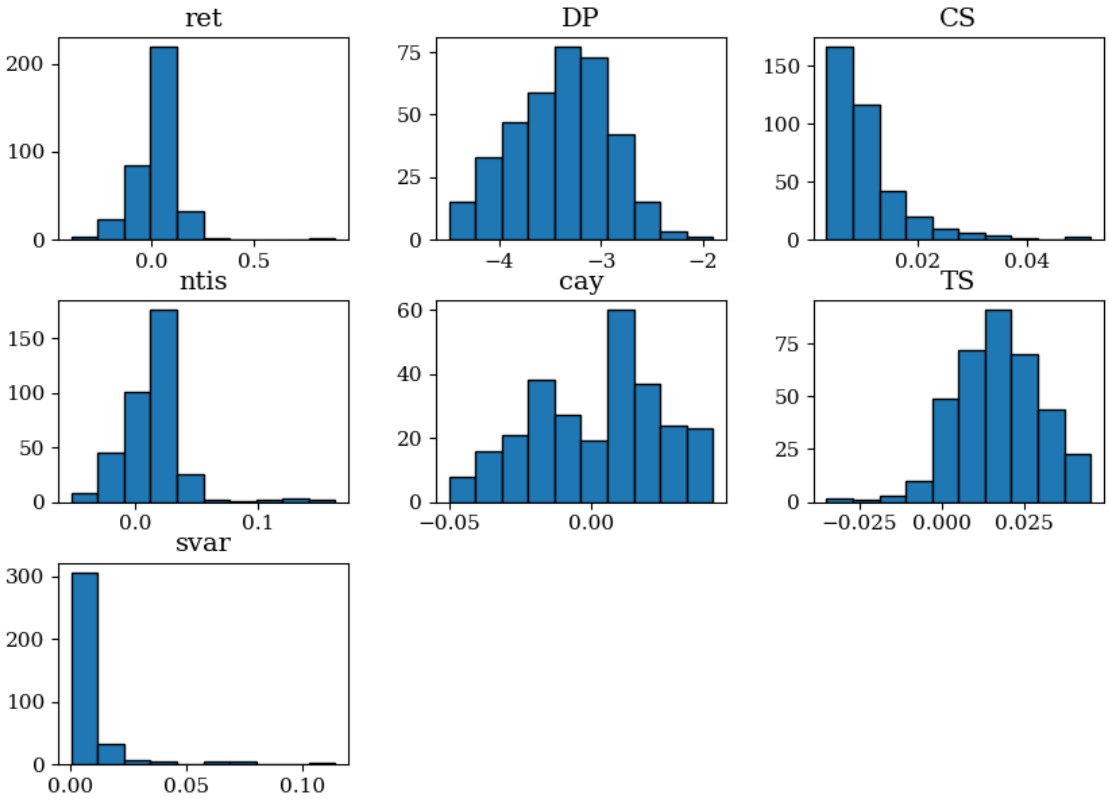
[10]:

	count	mean	std	min	25%	50%	75%	max
ret	365.0	0.027951	0.112073	-0.388075	-0.021158	0.035569	0.083677	0.893713
DP	365.0	-3.391056	0.470778	-4.493159	-3.797300	-3.373817	-3.042370	-1.903915
CS	365.0	0.010621	0.006545	0.003243	0.006502	0.008524	0.012306	0.051673
ntis	365.0	0.015432	0.025043	-0.051831	0.005041	0.016489	0.026695	0.163522
cay	273.0	0.001998	0.022772	-0.050141	-0.017083	0.007632	0.018796	0.042897
TS	365.0	0.017220	0.012820	-0.035000	0.009000	0.017500	0.026100	0.045300
svar	365.0	0.008814	0.015153	0.000370	0.002430	0.003984	0.007887	0.114436

[11]:
```
df.hist(figsize=(10,7), edgecolor='black', grid=False)
# Graphical Overview of the Distributions of Individual Variables in the Data
# The variable "date" is not included as it is not a numerical variable.
```

[11]:
```
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)
```



Question 1: Preparing and analyzing the data

1. First we need to align the data such that a row that contains the features for date t, contains the return for date $t+1$ (instead of the return for date t as it does now). This ensures that we are actually predicting **next** quarters returns. For this we need to lead the return time series by one period. (Hint: Use the `shift()` function to add a new variable to the dataframe that is the return led by one period. Afterwards remove the old return time series from the dataframe.)

[12]:
```
df['ret'] = df['ret'].shift(-1)  # Shifting the Return by the Time Period t+1.

df.tail()
```

[12]:

	date	ret	DP	CS	ntis	cay	TS	svar
360	2019-Q1	0.042688	-3.943400	0.010258	-0.023230	-0.039336	0.0017	0.004651
361	2019-Q2	0.017042	-3.960033	0.010006	-0.012562	-0.033844	-0.0010	0.003271
362	2019-Q3	0.090143	-3.951689	0.008505	-0.010862	-0.029529	-0.0019	0.005517
363	2019-Q4	-0.193794	-4.015896	0.008410	-0.007222	-0.033609	0.0032	0.002319
364	2020-Q1	NaN	-3.769992	0.012252	-0.007731	-0.050141	0.0058	0.079049

2. Remove all rows that contain missing values from the dataset. Google will provide many different ways on how to do this. If you struggle with this exercise, do the following: Use the `.isna().sum()` function to determine all rows that contain missing values. Find the missing values for these variables by eye inspection. Start and end the sample such that these rows with missing values are not included. Use the `.isna().sum()` again to make sure that you got rid of all missing values (`NaN`'s).

[13]:
```
df.isna().sum()  # Enumeration of all NaNs (per variable).
```

[13]:
```
date     0
ret      1
DP       0
CS       0
ntis     0
cay     92
TS       0
svar     0
dtype: int64
```

[14]:
```
df = df.dropna()  # Discarding all rows where variables have a NaN cell.

df.isna().sum()
```

[14]:
```
date    0
ret     0
DP      0
CS      0
ntis    0
cay     0
TS      0
svar    0
dtype: int64
```

3. Split the sample into two parts. Data before 1995 for *training* and *validation* and data after and including 1995 for *out-of-sample* testing.

[15]:
```
# Creating variables with the information "1994-Q4" and the position serving as␣
 ↪the intersection.
split_date = '1994-Q4'                                  # Variable with split␣
 ↪value
split_ind = df.index[df['date'] == split_date][0]      # Variable with split␣
 ↪position

# Division of the data into "train_data" and "test_data".
train_data = df.loc[:split_ind]                         # In-sample dataset (all␣
 ↪rows up to the split position)
test_data = df.loc[split_ind + 1:]                      # Out-of-sample dataset␣
 ↪(all rows after the split position)

print(f"train_data contains {len(train_data)} observations.")
print(f"test_data contains {len(test_data)} observations.")
```

```
train_data contains 172 observations.
test_data contains 100 observations.
```

4. Compute the mean quarterly return and its standard deviation in the training and test data. Is there anything worth noting?

[16]:
```
train_mean_ret = train_data['ret'].mean()     # Average quarterly return␣
 ↪(train_data)
train_std_ret = train_data['ret'].std()       # Standard deviation of quarterly␣
 ↪return (train_data)

test_mean_ret = test_data['ret'].mean()       # Average quarterly return␣
 ↪(test_data)
test_std_ret = test_data['ret'].std()         # Standard deviation of quarterly␣
 ↪return (test_data)

# Output of the results
print("Train data:")
print(f"Average quarterly return: {train_mean_ret:.4f}")
```

```
print(f"Standard deviation of quarterly return: {train_std_ret:.4f}")
print("\nTest data:")
print(f"Average quarterly return: {test_mean_ret:.4f}")
print(f"Standard deviation of quarterly return: {test_std_ret:.4f}")
```

```
Train data:
Average quarterly return: 0.0306
Standard deviation of quarterly return: 0.0763

Test data:
Average quarterly return: 0.0252
Standard deviation of quarterly return: 0.0823
```

5. Compute the correlation matrix for the training data (including both the outcomes and the features). Is there anything worth noting?

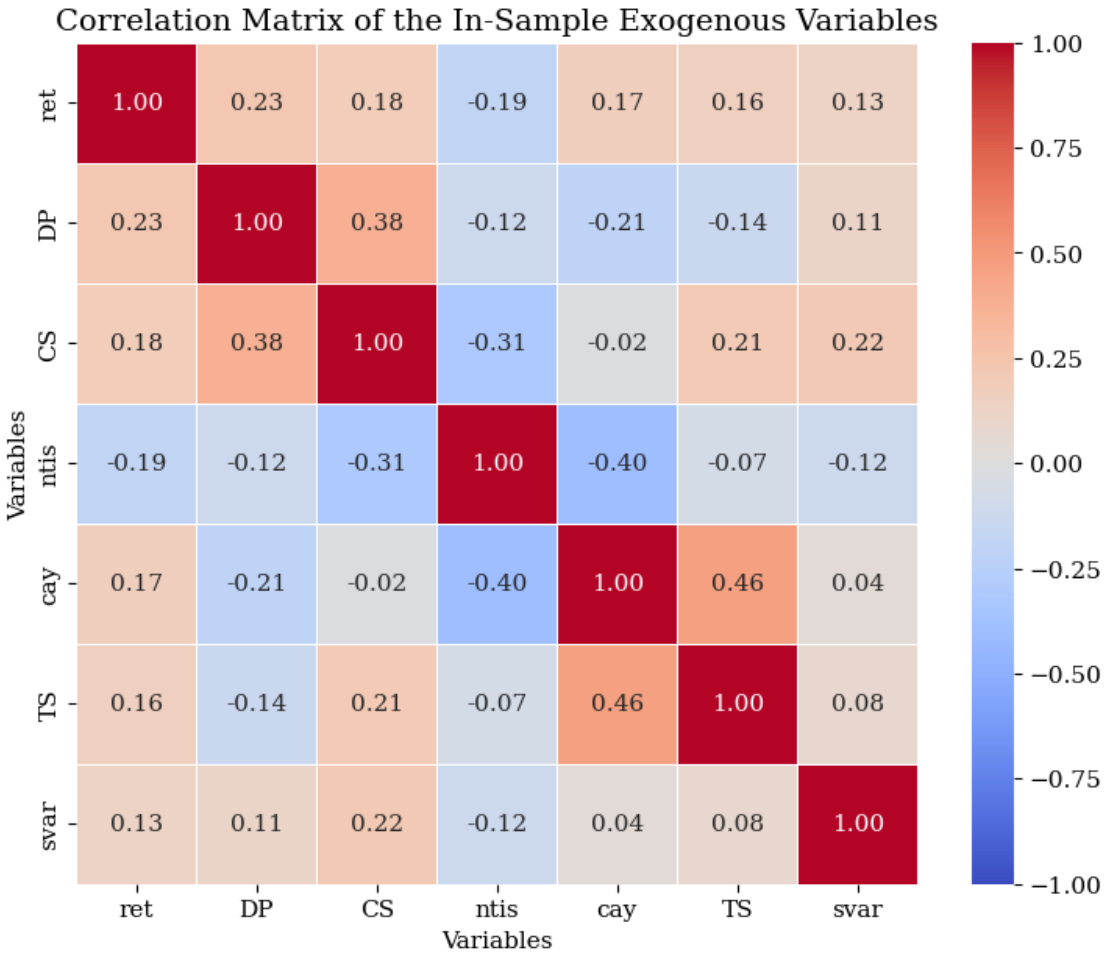
[17]:
```
# Calculation of the correlation matrix for the training data

# "date" column excluded (Only numerical columns)
train_cor_matrix = train_data.loc[:, train_data.columns != 'date'].
 ↪corr(method='pearson')

# Output of the correlation matrix. Values rounded to two decimal places.
print("Correlation matrix (train data):")
print(round(train_cor_matrix,2))
```

```
Correlation matrix (train data):
       ret    DP    CS  ntis   cay    TS  svar
ret   1.00  0.23  0.18 -0.19  0.17  0.16  0.13
DP    0.23  1.00  0.38 -0.12 -0.21 -0.14  0.11
CS    0.18  0.38  1.00 -0.31 -0.02  0.21  0.22
ntis -0.19 -0.12 -0.31  1.00 -0.40 -0.07 -0.12
cay   0.17 -0.21 -0.02 -0.40  1.00  0.46  0.04
TS    0.16 -0.14  0.21 -0.07  0.46  1.00  0.08
svar  0.13  0.11  0.22 -0.12  0.04  0.08  1.00
```

Graphical representation of the correlation matrix (extra)

[18]:
```
fig, ax = plt.subplots(figsize=(9, 7))
s = sns.heatmap(train_data.loc[:, train_data.columns != 'date'].corr(),
                annot=True,
                center=0,
                linewidths=.5,
                square=True,
                vmin=-1,
                vmax=1,
                xticklabels='auto', # automatic X-variables
                yticklabels='auto', # automatic X-variables
```

```
                fmt='0.2f',
                cmap="coolwarm")
s.set_title('Correlation Matrix of the In-Sample Exogenous Variables')
s.set(xlabel='Variables', ylabel='Variables')
plt.show()
```



Question 2: Predicting returns

After having cleaned the data, you are ready to build the first model to predict returns

1. Use the training data to fit a linear model using all features (make sure to exclude the date variable). Which features are useful for predicting returns?

Regression with all predictors: `ret ~ .`

Solution Approach 1: `ols()` "formula" function from `statsmodels.formula`

- Advantage:
  - Formula function, as known from R.
    - **Extract**:
    - `np.log(Variable)` Apply logarithm to Variable
    - `np.sqrt(Variable)` Apply square root to Variable
    - `np.power(Variable, 2)` Variable to the power of 2 Example: `np.power(Variable, 3)` Variable to the power of 3
    - `C(Variable)` The C declares the Variable as a categorical variable
    - `Variable1 * Variable2` Interaction term of two variables (with additional inclusion of individual variables)
    - `Variable1 : Variable2` Interaction term of two variables (without separately including individual variables)
  - Provides a summary statistic directly.
- Disadvantage:
  - Often not flexible enough for more complex tasks…

[19]:
```
# Fit in-sample multilinear regression
Fit_lm = ols('ret ~ DP + CS + ntis + cay + TS + svar', data=train_data).fit()
Fit_lm.summary()   # Outputting the model statistics.

# Option 2:
## Concatenate all predictor variable names except 'ret'
#predictors = [col for col in Auto.columns if col != 'ret']
#formula = 'ret ~ ' + ' + '.join(predictors)

## Fit linear regression using all predictors from Auto data
#Fit_lm = ols(formula=formula, data=Auto).fit()
#Fit_lm.summary()   # Outputting the model statistics.
```

[19]:

Dep. Variable:	ret	**R-squared:**	0.129
Model:	OLS	**Adj. R-squared:**	0.097
Method:	Least Squares	**F-statistic:**	4.063
Date:	Tue, 01 Apr 2025	**Prob (F-statistic):**	0.000793
Time:	21:31:56	**Log-Likelihood:**	210.80
No. Observations:	172	**AIC:**	-407.6
Df Residuals:	165	**BIC:**	-385.6
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P> |t|	[0.025	0.975]
Intercept	0.2968	0.097	3.063	0.003	0.106	0.488
DP	0.0839	0.028	3.031	0.003	0.029	0.139
CS	0.4750	1.739	0.273	0.785	-2.958	3.908
ntis	-0.3945	0.410	-0.961	0.338	-1.205	0.416
cay	0.4215	0.306	1.379	0.170	-0.182	1.025
TS	0.6310	0.482	1.309	0.192	-0.320	1.583
svar	0.8027	0.828	0.969	0.334	-0.832	2.438

Omnibus:	26.211	**Durbin-Watson:**	1.810
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	40.178
Skew:	-0.823	**Prob(JB):**	1.89e-09
Kurtosis:	4.701	**Cond. No.**	1.10e+03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.1e+03. This might indicate that there are
strong multicollinearity or other numerical problems.

Solution Approach 2: `OLS()` without the "formula" function from `statsmodels`

- Advantage:
  - Provides a summary statistic directly.
- Disadvantage:
  - An "Intercept" must be added independently to the list of exogenous variables.
  - Often not flexible enough for more complex tasks…

[20]:
```
# Creating the model matrix:

# Removing the endogenous and qualitative variables.
X = train_data.drop(columns=['date', 'ret'])
X = sm.add_constant(X)
# Option 2: Inserting an intercept term at the beginning of the matrix.
#X.insert(0, 'intercept', np.ones(train_data.shape[0]))

y = train_data['ret']                          # Setting the endogenous variable.

model = sm.OLS(y, X)                           # Filling the Ordinary Least␣
 ↪Squares (OLS) model.
fit_lm = model.fit()                           # Fitting the univariate linear␣
 ↪regression model.
fit_lm.summary()                               # Outputting the model statistics.
```
[20]:

Dep. Variable:	ret	**R-squared:**	0.129
Model:	OLS	**Adj. R-squared:**	0.097
Method:	Least Squares	**F-statistic:**	4.063
Date:	Tue, 01 Apr 2025	**Prob (F-statistic):**	0.000793
Time:	21:31:56	**Log-Likelihood:**	210.80
No. Observations:	172	**AIC:**	-407.6
Df Residuals:	165	**BIC:**	-385.6
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P> |t|	[0.025	0.975]
const	0.2968	0.097	3.063	0.003	0.106	0.488
DP	0.0839	0.028	3.031	0.003	0.029	0.139
CS	0.4750	1.739	0.273	0.785	-2.958	3.908
ntis	-0.3945	0.410	-0.961	0.338	-1.205	0.416
cay	0.4215	0.306	1.379	0.170	-0.182	1.025
TS	0.6310	0.482	1.309	0.192	-0.320	1.583
svar	0.8027	0.828	0.969	0.334	-0.832	2.438

Omnibus:	26.211	**Durbin-Watson:**	1.810
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	40.178
Skew:	-0.823	**Prob(JB):**	1.89e-09
Kurtosis:	4.701	**Cond. No.**	1.10e+03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.1e+03. This might indicate that there are strong multicollinearity or other numerical problems.

Solution Approach 3: `LinearRegression()` without the "formula" function from `sklearn.linear_model`

- Advantage:
  - No need to add an "Intercept" to the list of exogenous variables.
  - Comprehensive and flexible enough for more complex tasks...
- Disadvantage:
  - Does **not** provide "out of the box" summary statistics.

For solving the following tasks, LinearRegression() is the best choice. Proceeding with Approach 3 from here.

[21]:
```
# Establishing y and X based on the training and validation datasets.
X_train = train_data.drop(columns=['ret','date'])   # Exogenous: all variables␣
 ↪except ret and date.
y_train = train_data['ret']                         # Endogenous: ret.
X_test = test_data.drop(columns=['ret','date'])     # Exogenous: all variables␣
 ↪except ret and date.
y_test = test_data['ret']                           # Endogenous: ret.
```

```
# Setting up and training the regression model.
model_all = LinearRegression()
model_all.fit(X_train, y_train)
```

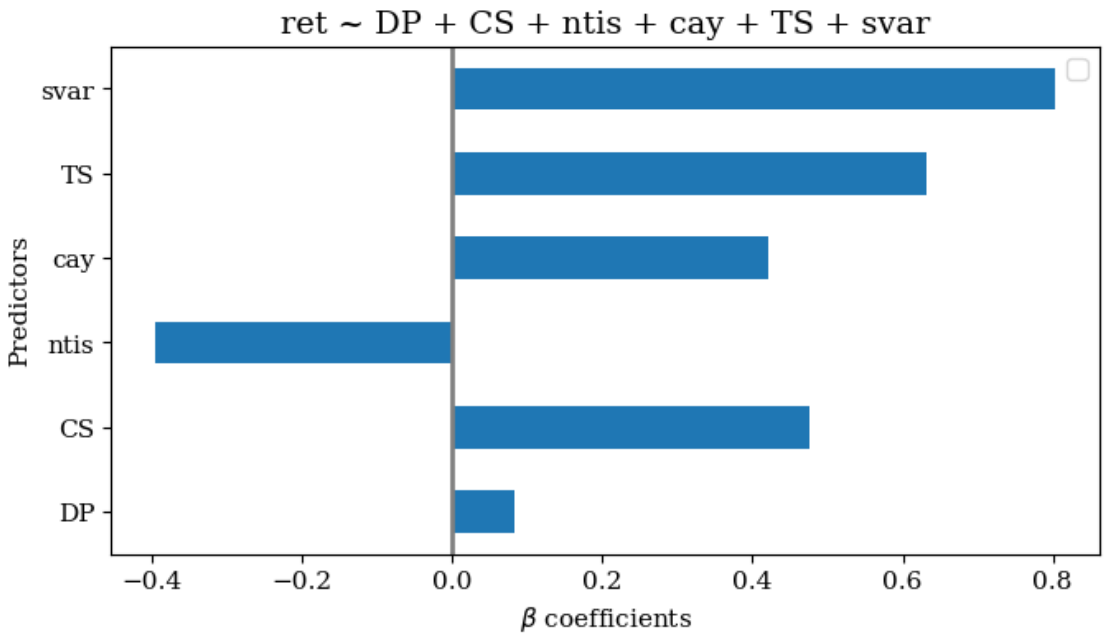
[21]:
```
LinearRegression()
```

[22]:
```
Feature_names = model_all.feature_names_in_
COefs = pd.DataFrame(
    model_all.coef_,              # Coeffocient values
    columns=["Coefficients"],     # New Column name
    index=Feature_names,          # Predictor name list
)

COefs.iloc[::-1]  # ".iloc[::-1]" reverses the order of the dataframe
```

[22]:

	Coefficients
svar	0.802650
TS	0.631040
cay	0.421456
ntis	-0.394479
CS	0.474950
DP	0.083873

[23]:
```
COefs.plot.barh(figsize=(10, 4))
plt.title("ret ~ DP + CS + ntis + cay + TS + svar")
plt.axvline(x=0, color=".5")
plt.xlabel("$\\beta$ coefficients")
plt.ylabel("Predictors")
#plt.legend(["zero","$\\beta$"])
plt.legend([])  # Remove legend
plt.subplots_adjust(left=0.3)
```



Relevance and Effect Size of Variables for the Model (extra)

[24]:
```
from sklearn.model_selection import train_test_split
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import Ridge
from sklearn.compose import TransformedTargetRegressor
```

[25]:
```
# Initialize and train the Ridge regression model
# Alpha parameter controls regularization strength (1=ridge)
ridge_model = Ridge(alpha=1e-10, random_state=1)
ridge_model.fit(X_train, y_train)

# Optionally, you can perform cross-validation to evaluate the model
mse_cv = -cross_val_score(ridge_model, X_train, y_train, cv=5,␣
 ↪scoring='neg_mean_squared_error').mean()
print("5-fold cross-validated MSE:", mse_cv)
```

```
5-fold cross-validated MSE: 0.008093899166081891
```

[26]:
```
X = train_data.drop(columns=['ret','date'])    # Exogenous: all variables except␣
 ↪ret and date.
y = train_data['ret']                          # Endogenous: ret.
```

[27]:
```
# Setting up the train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

[28]:
```
train_dataset = X_train.copy()
train_dataset.insert(0, "ret", y_train)
```

In the following section, we will interpret the coefficients of the model. While we do so, we should keep in mind that any conclusion we draw is about the model that we build, rather than about the true (real-world) generative process of the data.

[29]:
```
categorical_columns = []
numerical_columns = ['DP', 'CS', 'ntis', 'cay', 'TS', 'svar']

preprocessor = make_column_transformer(
    (OneHotEncoder(), categorical_columns),
    remainder="passthrough",
    verbose_feature_names_out=False,  # avoid to prepend the preprocessor names
    force_int_remainder_cols=False    # Enable the future behavior for␣
 ↪remainder columns
)

# Create a model pipeline
model = make_pipeline(
    preprocessor,
    TransformedTargetRegressor(
        regressor=Ridge(alpha=1e-10),  # Ridge regression model
        func=np.log1p,                 # Apply log transformation to target␣
 ↪variable
        inverse_func=np.expm1          # Inverse of log transformation for␣
 ↪predictions
    )
)

# Fit the model
model.fit(X_train, y_train)
```

[29]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(force_int_remainder_cols=False,
                                   remainder='passthrough',
                                   transformers=[('onehotencoder',
                                                  OneHotEncoder(), [])],
                                   verbose_feature_names_out=False)),
                ('transformedtargetregressor',
                 TransformedTargetRegressor(func=<ufunc 'log1p'>,
                                            inverse_func=<ufunc 'expm1'>,
                                            regressor=Ridge(alpha=1e-10)))])
```

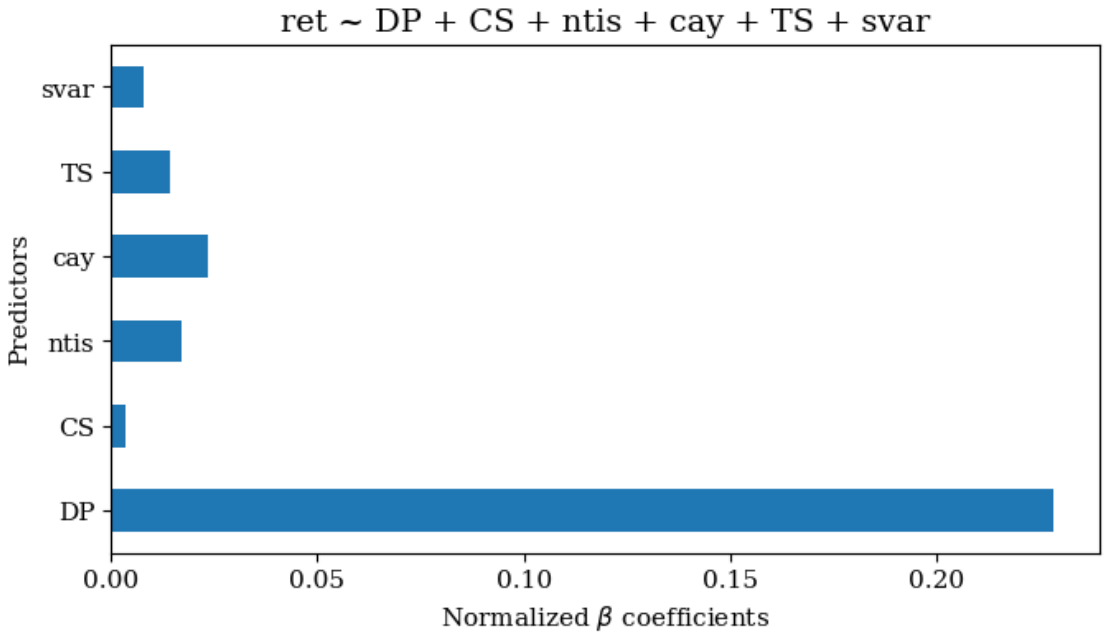
Normalization with the Z-Score

The Z-score (or normalized value) is a standardized value with a mean of 0 and a standard deviation of 1. It is calculated by subtracting the value from the sample mean and then dividing by the sample standard deviation.

$$z = \frac{(x - Mean)}{Standard\ Deviation}$$

When applied to numerical values in a vector, it results in a standardized vector.

[30]:
```
feature_names = model[:-1].get_feature_names_out()

X_train_preprocessed = pd.DataFrame(model[:-1].transform(X_train),␣
 ↪columns=feature_names, copy=None)

X_train_preprocessed.std(axis=0).plot.barh(figsize=(10, 4))
plt.title("ret ~ DP + CS + ntis + cay + TS + svar")
plt.xlabel("Normalized $\\beta$ coefficients")
plt.ylabel("Predictors")
plt.subplots_adjust(left=0.3)
```

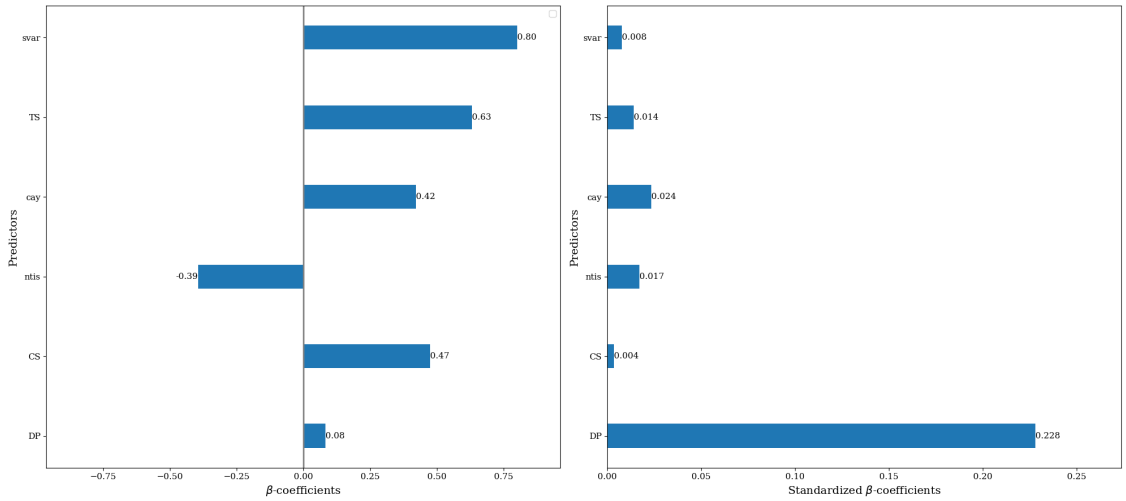


[31]:
```
X_train_preprocessed.std(axis=0)
```

[31]:
```
DP      0.228184
CS      0.003674
ntis    0.017305
cay     0.023599
TS      0.014240
svar    0.007801
dtype: float64
```

[32]:
```
# Create a figure with two subplots arranged horizontally

fig, axes = plt.subplots(1, 2, figsize=(20, 9))  # 1 row, 2 columns
#fig.suptitle('ret ~ DP + CS + ntis + cay + TS + svar', fontsize=12,␣
 ↪fontweight='bold')

# Plot the first horizontal bar plot
COefs.plot.barh(ax=axes[0], width=0.3)
#axes[0].set_title("Regression Coefficients' Magnitudes")
axes[0].axvline(x=0, color=".5")
axes[0].set_xlabel("$\\beta$-coefficients", fontsize=14)
axes[0].set_ylabel("Predictors", fontsize=14)
axes[0].legend([])  # Remove legend

# Place the beta values next to the horizontal bars
for i, (v, p) in enumerate(zip(COefs.values[:, 0], COefs.index)):
    if v < 0:  # If negative coefficient, place to the left of the bar
        axes[0].text(v, i, f'{v:.2f}', va='center', ha='right', fontsize=11,␣
 ↪color='black', weight='normal')
    else:
        axes[0].text(v, i, f'{v:.2f}', va='center', ha='left', fontsize=11,␣
 ↪color='black', weight='normal')

# Adjust x-axis limits to provide space for negative coefficient values
max_abs_coef = abs(COefs.values[:, 0]).max()
axes[0].set_xlim(-max_abs_coef * 1.2, max_abs_coef * 1.2)

# Plot the second horizontal bar plot
sorted_featureWeight = (X_train_preprocessed.std(axis=0)).
 ↪sort_values(ascending=True)
#sorted_featureWeight.plot.barh(ax=axes[1])
X_train_preprocessed.std(axis=0).plot.barh(ax=axes[1], width=0.3)
#axes[1].set_title("Normalized Variables: Z-scores")
# Standardized betas (beta coefficients) are normalized based on the scales of␣
 ↪the predictors.
axes[1].set_xlabel("Standardized $\\beta$-coefficients", fontsize=14)
axes[1].set_ylabel("Predictors", fontsize=14)

# Place the beta values next to the horizontal bars for the second plot
for i, (v, p) in enumerate(zip(X_train_preprocessed.std(axis=0),␣
 ↪X_train_preprocessed.columns)):
    if v < 0:  # If negative coefficient, place to the left of the bar
        axes[1].text(v, i, f'{v:.3f}', va='center', ha='right', fontsize=11,␣
 ↪color='black', weight='normal')
    else:
```

```
        axes[1].text(v, i, f'{v:.3f}', va='center', ha='left', fontsize=11,␣
 ↪color='black', weight='normal')

# Adjust x-axis limits for the second plot
max_abs_value = abs(X_train_preprocessed.std(axis=0)).max()
axes[1].set_xlim(0, max_abs_value * 1.2)

# Adjust spacing between subplots
plt.subplots_adjust(wspace=0.2)
# Tighten the layout to remove oversized margins
plt.tight_layout()
# Save the figure
#plt.savefig("2.4.beta_coef_results.png")

#![This plot shows that...](2.4.beta_coef_results.png)
#plt.show()
```



2. Compute the in-sample R^2 as well as the mean squared error. Do you think quarterly returns can easily be predicted?

[33]:
```
y_pred_train = model_all.predict(X_train)                    # In-sample␣
 ↪predictions.

r2_train = r2_score(y_train, y_pred_train)                   # Option 1:␣
 ↪calling sklearn r2_score function
print(f"In-sample (all) R^2: {r2_train:.4f}")

#model_r2_train = model_all.score(X_train, y_train)          # Option 2:␣
 ↪calling LinearRegression.score function
#print(f"In-sample R^2: {model_r2_train:.4f}")
```

```
mse_train = mean_squared_error(y_train, y_pred_train)          # In-sample MSE
print(f"In-sample (all) MSE: {mse_train:.5f}")
```

```
In-sample (all) R^2: 0.1321
In-sample (all) MSE: 0.00441
```

3. Use 5-fold cross validation to obtain an estimate for the out-of-sample mean squared error. Compare this estimate to the in-sample mean squared error from *Q2.2*.

[34]:
```
# Perform 5-fold cross-validation and calculate the in-sample MSE
train_all_cv_mse = -cross_val_score(model_all, X_train, y_train,␣
 ↪scoring='neg_mean_squared_error',
                                    cv=KFold(n_splits=5, shuffle=True,␣
 ↪random_state=1)).mean()
print(f"In-sample (all) cv.MSE: {train_all_cv_mse:.5f}")
```

```
In-sample (all) cv.MSE: 0.00521
```

4. Based on your findings from *Q2.1* select only a subset of the features to improve your model. Which features do you choose and why? Compute the in-sample R^2 as well as the mean squared error for this model and use 5-fold cross validation to obtain an estimate for the out-of-sample mean squared error. Compare your results to the model using all features.

Regression with DP as predictor: `ret ~ DP`

[53]:
```
# Setting up a new regression model.
# Pre-selecting the exogenous variable.
selected_features1 = ['DP']

# Establishing y and X based on the training and validation datasets.
X_train_DP_selected = train_data[selected_features1]
X_test_DP_selected = test_data[selected_features1]
y_train_DP = train_data['ret']
y_test_DP = test_data['ret']

# Training the regression model.
model_DP = LinearRegression()
model_DP.fit(X_train_DP_selected, y_train_DP)

# In-sample predictions.
y_pred_train_DP_selected = model_DP.predict(X_train_DP_selected)

# Determining the in-sample R^2.
r2_in_sample_DP_selected = r2_score(y_train_DP, y_pred_train_DP_selected)
print(f"In-sample (DP) R^2: {r2_in_sample_DP_selected:.4f}")

# Determining the in-sample Mean Squared Error (MSE).
mse_train_DP_selected = mean_squared_error(y_train_DP, y_pred_train_DP_selected)
```

```
print(f"In-sample (DP) MSE: {mse_train_DP_selected:.4f}")

# Perform 5-fold cross-validation and calculate the in-sample MSE
train_DP_cv_mse = -cross_val_score(model_DP,
                                   X_train_DP_selected,
                                   y_train_DP,
                                   scoring='neg_mean_squared_error',
                                   cv=KFold(n_splits=5, shuffle=True,␣
 ↪random_state=1)).mean()

print(f"In-sample (DP) cv.MSE: {train_DP_cv_mse:.5f}")
```

```
In-sample (DP) R^2: 0.0541
In-sample (DP) MSE: 0.0055
In-sample (DP) cv.MSE: 0.00556
```

Regression with DP and cay as predictors: `ret ~ DP + cay`

[54]:
```
# Setting up a new regression model.
# Pre-selecting the exogenous variable.
selected_features2 = ['DP', 'cay']

# Establishing y and X based on the training and validation datasets.
X_train_DPcay_selected = train_data[selected_features2]
X_test_DPcay_selected = test_data[selected_features2]
y_train_DPcay = train_data['ret']
y_test_DPcay = test_data['ret']

# Training the regression model.
model_DPcay = LinearRegression()
model_DPcay.fit(X_train_DPcay_selected, y_train_DPcay)

# In-sample predictions.
y_pred_train_DPcay_selected = model_DPcay.predict(X_train_DPcay_selected)

# Determining the in-sample R^2.
r2_in_sample_DPcay_selected = r2_score(y_train_DPcay,␣
 ↪y_pred_train_DPcay_selected)
print(f"In-sample (DP+cay) R^2: {r2_in_sample_DPcay_selected:.4f}")

# Determining the in-sample Mean Squared Error (MSE).
mse_train_DPcay_selected = mean_squared_error(y_train_DPcay,␣
 ↪y_pred_train_DPcay_selected)
print(f"In-sample (DP+cay) MSE: {mse_train_DPcay_selected:.4f}")

# Perform 5-fold cross-validation and calculate the in-sample MSE
train_DPcay_cv_mse = -cross_val_score(model_DPcay,
```

```
                                        X_train_DPcay_selected,
                                        y_train_DPcay,
                                        scoring='neg_mean_squared_error',
                                        cv=KFold(n_splits=5, shuffle=True,␣
 ↪random_state=1)).mean()

print(f"In-sample (DP+cay) cv.MSE: {train_DPcay_cv_mse:.5f}")
```

```
In-sample (DP+cay) R^2: 0.1040
In-sample (DP+cay) MSE: 0.0052
In-sample (DP+cay) cv.MSE: 0.00529
```

5. Now suppose you use the two models you have build to predict quarterly returns in the coming 25 years. Compute the out-of-sample mean squared errors for the test data. Compare these errors to the estimates for the out-of-sample errors obtained from k-fold CV. Interpret.

[55]:
```
## Perform 5-fold cross-validation - calculate the out-of-sample (all) MSE:
test_all_cv_mse = -cross_val_score(model_all,
                                   X_test,
                                   y_test,
                                   scoring='neg_mean_squared_error',
                                   cv=KFold(n_splits=5, shuffle=True,␣
 ↪random_state=1)).mean()


## Perform 5-fold cross-validation - calculate the out-of-sample (DP) MSE:
test_DP_cv_mse = -cross_val_score(model_DP,
                                  X_test_DP_selected,
                                  y_test_DP,
                                  scoring='neg_mean_squared_error',
                                  cv=KFold(n_splits=5, shuffle=True,␣
 ↪random_state=1)).mean()


## Perform 5-fold cross-validation - calculate the out-of-sample (DP+cay) MSE:
test_DPcay_cv_mse = -cross_val_score(model_DPcay,
                                     X_test_DPcay_selected,
                                     y_test_DPcay,
                                     scoring='neg_mean_squared_error',
                                     cv=KFold(n_splits=5, shuffle=True,␣
 ↪random_state=1)).mean()
```

[61]:
```
# Define the data
model_data_table = {
    "Model": ['All predictors', 'DP', 'DP+cay'],
    "In-sample MSE": [mse_train, mse_train_DP_selected,␣
 ↪mse_train_DPcay_selected],
    "In-sample cv.MSE": [train_all_cv_mse, train_DP_cv_mse, train_DPcay_cv_mse],
```

```
    "Out-of-sample cv.MSE": [test_all_cv_mse, test_DP_cv_mse, test_DPcay_cv_mse]
}
# Create a DataFrame
mdt = pd.DataFrame(model_data_table)

# Display the DataFrame
print(mdt)
```

```
            Model  In-sample MSE  In-sample cv.MSE  Out-of-sample cv.MSE
0  All predictors       0.004414          0.005213              0.008361
1              DP       0.005479          0.005560              0.006735
2          DP+cay       0.005190          0.005294              0.006587
```

Question 3: Predicting the direction of the stock market

Instead of quantitatively predicting returns, assume now that you want to predict the direction of the stock market, that is, whether stocks go up or down. Based on these predictions you want to either invest in stocks or not.

1. Create a new variable in both, the training and test data that is 1 if the return is larger than zero and 0 otherwise.

[38]:
```
train_data_class = train_data.copy()  # Continuing with the copy of "train_data"
test_data_class = test_data.copy()    # Continuing with the copy of "test_data"

# Overwriting ret column.
# ret ist 1, wenn die Rendite größer als Null ist, sont 0.
train_data_class['ret'] = train_data_class['ret'].apply(lambda x: 1 if x > 0␣
 ↪else 0)  # Applying onto In-sample data
test_data_class['ret'] = test_data_class['ret'].apply(lambda x: 1 if x > 0 else␣
 ↪0)    # Applying onto Out-of-sample data

train_data_class.head()
```

[38]:

	date	ret	DP	CS	ntis	cay	TS	svar
92	1952-Q1	1	-2.842696	0.005328	0.032094	-0.010595	0.0104	0.002102
93	1952-Q2	0	-2.845711	0.005425	0.027731	0.000055	0.0089	0.001660
94	1952-Q3	1	-2.828741	0.005521	0.031038	-0.000695	0.0106	0.001076
95	1952-Q4	0	-2.936193	0.005231	0.026535	-0.015950	0.0070	0.001753
96	1953-Q1	0	-2.886819	0.004354	0.024013	-0.019021	0.0093	0.001574

2. Compute the proportion of positive stock returns in both, the training and test data.

[39]:
```
# In-sample proportion of positive stock returns
positive_proportion_train = train_data_class['ret'].mean()
print(f"In-sample proportion of positive stock returns:␣
 ↪{positive_proportion_train:.3f}")

# Out-of-Sample proportion of positive stock returns
positive_proportion_test = test_data_class['ret'].mean()
print(f"Out-of-Sample proportion of positive stock returns:␣
 ↪{positive_proportion_test:.3f}")
```

```
In-sample proportion of positive stock returns: 0.686
Out-of-Sample proportion of positive stock returns: 0.730
```

3. Fit a logistic regression using the training data to predict the direction of the stock market (make sure to exclude the *date* variable and the old quantitative *ret* variable.). Which features are useful predictors? Compute the in-sample accuracy and error rate. Do you think you have build a good model to predict the direction of the stock market?

[40]:
```
# Aufsetzen eines neuen Regressionsmodells
```

```
X_train_class = train_data_class[['TS', 'svar']]                    #␣
 ↪In-sample X
y_train_class = train_data_class['ret']                             #␣
 ↪In-sample y

# Regressionsmodell trainieren
model = LogisticRegression()
model.fit(X_train_class, y_train_class)

y_pred_train_class = model.predict(X_train_class)                   #␣
 ↪In-sample Schätzungen
probs_train_class = model.predict_proba(X_train_class)[:, 1]        #␣
 ↪Wahrscheinlichkeiten der vorhergesagten In-sample Schätzungen

pred_train_class = np.zeros(len(train_data_class))
pred_train_class[probs_train_class > 0.5] = 1                       #␣
 ↪Compute predictions using a threshold of 50%
                                                                    # Extra:
 ↪ Ab threshold ~70% flippen "accuracy" und "error rate" Werte.

print("Train data:")
accuracy_train_class = accuracy_score(y_train_class, pred_train_class)  #␣
 ↪In-sample accuracy
print(f"In-sample accuracy: {accuracy_train_class:.4f}")

error_rate_train_class = np.mean(pred_train_class != y_train_class)     #␣
 ↪In-sample error rate
print(f"In-sample error rate: {error_rate_train_class:.4f}")
```

```
Train data:
In-sample accuracy: 0.6860
In-sample error rate: 0.3140
```

4. Now suppose you use the model you have build to predict the direction of the stock market in the coming 25 years. Compute the out-of-sample accuracy and error rate for the test data. Compare these outcomes to the in-sample statistics. Do you think your model work well out-of-sample? Interpret the results.

[41]:
```
X_test_class = test_data_class[['TS', 'svar']]                      #␣
 ↪Out-of-sample X
y_test_class = test_data_class['ret']                               #␣
 ↪Out-of-sample y

y_pred_test_class = model.predict(X_test_class)                     #␣
 ↪Out-of-sample Schätzungen
probs_test_class = model.predict_proba(X_test_class)[:, 1]          #␣
 ↪Wahrscheinlichkeiten der vorhergesagten In-sample Schätzungen
```

```
pred_test_class = np.zeros(len(test_data_class))
pred_test_class[probs_test_class > 0.5] = 1                                  # Compute␣
 ↪predictions using a threshold of 50%

print("Validation data:")
accuracy_test_class = accuracy_score(y_test_class, pred_test_class)    #␣
 ↪Out-of-sample accuracy
print(f"Out-of-sample accuracy: {accuracy_test_class:.4f}")

error_rate_test_class = np.mean(pred_test_class != y_test_class)       #␣
 ↪Out-of-sample error rate
print(f"Out-of-sample error rate: {error_rate_test_class:.4f}")
```

```
Validation data:
Out-of-sample accuracy: 0.7300
Out-of-sample error rate: 0.2700
```

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.