Predictive Modeling

Series 4

Exercise 4.1

The data file **fitness.csv** contains measurements of a fitness test of 31 patients. The response variable **oxy** refers to the rate of oxygen consumption which was measured by means of a complicated and expensive procedure. Predictors are **age**, **weight**, **runtime** (running time), **rstpulse** (resting pulse), **runpulse** (running/active pulse) and **maxpulse** (maximal pulse).

a) Load the data file **fitness.csv** and fit a regression model that contains all predictors. Is there any association between predictors and response variable?

   Python code:

```
[ ]: import pandas as pd

     # Load data
     fitness = pd.read_csv('./data/fitness.csv')

     # As a first inspection, print the first rows of the data:
     print(fitness.head())
     # As well as the dimensions of the set:
     print('\nSize of fitness =\n', fitness.shape)
```

b) Verify whether variable transformations are necessary by carrying out a residual analysis. You can use the same **Python**-functions as were used in the previous exercise series.

   If necessary, adapt the model so that it does not show any systematic errors or any other violations of the model assumptions.

   Optional, for the advanced reader: By using *partial residual plots* you can check whether all predictors have been included in the correct form. How do you interpret partial residual plots?

c) Analyze the data with respect to pairwise correlations of the predictors.

   Python-Hint:

[ ]:
```python
import seaborn as sns

fig = sns.pairplot(x)
plt.show()
```

d) Check whether there is high multicollinearity by computing the VIFs.

`Python`-**Hint**:

[ ]:
```python
""" Hints: """
import numpy as np
import statsmodels.api as sm
from statsmodels.stats.outliers_influence \
import variance_inflation_factor

# Unfortunately the Method variance_inflation_factor() contains
# an error Therefore we make the analysis ourselves:
def VIF_analysis(x):
    """ VIF analysis of variables saved in x
    Input:
    x: m*n matrix or Dataframe, containing m samples of n predictors
    Output:
    VIF: Vector containing n Variance Inflation Factors
    """
    # Preproces:
    x_np = x.to_numpy()
    VIF = []
    # For all n Predictors:
    for i in range(x.shape[1]):
        # Define x and y
        x_i = np.delete(x_np, i, 1)
        x_i = sm.add_constant(x_i)
        y_i = x_np[:, i]
        # Fit model
        model = sm.OLS(y_i, x_i).fit()
        # Calculate the VIF
        VIF.append(1 / (1 - model.rsquared))
    return VIF
```

e) Alleviate the multicollinearity problem by using different methods:

i) Amputation, i.e. leave out redundant variables.

ii) Create new variables that are not collinear, e.g. by considering the quotient of two correlated variables.

Save the fitted values for each of these adapted models and carry out a pairwise comparison by generating pairwise scatterplots. `Python`-**Hints**:

```
[ ]: pred_i = model_i.predict(sm.add_constant(x_i))
     pred_ii = model_ii.predict(sm.add_constant(x_ii))


     fig = plt.figure(figsize = (6,5))
     ax1 = fig.add_subplot(1, 1, 1)

     plot_reg(ax1, pred_i, pred_ii,
              x_lab="pred_i", y_lab="pred_ii",
              title="Comparing predictions")
     plt.show()
```

What do you observe?

f) Use the model from part e) (ii) and perform a forward stepwise model selection using the AIC as criterion in order to reduce the set of predictors. Consider an ANOVA-test between the model from part e) (ii) and the reduced model to decide whether the omitted variables are indeed redundant. `Python`-**Hints**:

```
[ ]: def fit_linear_reg(x, y):
         '''Fit Linear model with predictors x on y
         return AIC, BIC, R2 and R2 adjusted '''
         x = sm.add_constant(x)
         # Create and fit model
         model_k = sm.OLS(y, x).fit()

         # Find scores
         BIC = model_k.bic
         AIC = model_k.aic
         R2 = model_k.rsquared
         R2_adj = model_k.rsquared_adj
         RSS = model_k.ssr

         # Return result in Series
         results = pd.Series(data={'BIC': BIC, 'AIC': AIC, 'R2': R2,
                                   'R2_adj': R2_adj, 'RSS': RSS})

         return results


     def add_one(x_full, x, y, scoreby='RSS'):
         ''' Add possible predictors from x_full to x,
         Fit a linear model on y using fit_linear_reg
         Returns Dataframe showing scores as well as best model '''
         # Predefine DataFrame
         x_labels = x_full.columns
         zeros = np.zeros(len(x_labels))
         results = pd.DataFrame(
             data={'Predictor': x_labels.values, 'BIC': zeros,
                    'AIC': zeros, 'R2': zeros,
                    'R2_adj': zeros, 'RSS': zeros})

         # For every predictor find R^2, RSS, and AIC
         for i in range(len(x_labels)):
             x_i = np.concatenate((x, [np.array(x_full[x_labels[i]])]))
             results.iloc[i, 1:] = fit_linear_reg(x_i.T, y)

         # Depending on where we scoreby, we select the highest or lowest
         if scoreby in ['RSS', 'AIC', 'BIC']:
             best = x_labels[results[scoreby].argmin()]
         elif scoreby in ['R2', 'R2_adj']:
             best = x_labels[results[scoreby].argmax()]

         return results, best
```

[ ]:
```
""" Python Hints: """
# Define the empty dataframe and the empty predictor
results = pd.DataFrame(data={'Best_Pred': [], 'AIC':[]})

# Define the empty predictor
x0 = [np.zeros(len(y))]

x_red = x.copy() # Copy the full set of predictors
x_forward = x0

for i in range(x_red.shape[1]):
    # Find results and best predictor using add_one()

    # Update the empty predictor with the best predictor,
    # using np.concatenate()

    # Remove the chosen predictor from the list of options,
    # using pd.drop()

    # Save results


print('Best Predictors and corresponding AIC:\n', results,
      '\n\nThe best model thus contains',
      results['AIC'].argmin() + 1, ' predictors')
```

g) The goal of this study is to substitute the expensive and complicated procedure required to measure the rate of oxygen consumption by a set of predictors that are inexpensive to measure. Is this possible, and if so, how would you do it?

Exercise 4.2

In a study about controlling infection risk in US hospitals, a random sample of 113 hospitals were considered along with 12 measured quantities. After loading the data file `senic.csv`, check whether `region` and `school` are recognized as a factor variables. If not, transform them into factor variables.

In **Python**, you can use `pandas.get_dummies()` to transform a predictor into dummy variables.

By using the following variables as predictors:

- `age` : average age of patients in years
- `inf` : average infection risk in percent
- `region` : geographical region with $1 = \text{NE}$, $2 = \text{N}$, $3 = \text{S}$ and $4 = \text{W}$
- `beds` : number of beds
- `pat` : average number of patients a day
- `nurs` : number of full-employed and trained nurses

and

- `length` : the average duration of hospital stay in days

as response variable, carry out a linear regression analysis and select an optimal model by following these instructions:

a) Check the correlations between these variables. Which of them are problematic and why? Is there an intuitive explanation of this problem?

   Alternatively to the function `seaborn.pairplot()`, you can also find the correlations using `pandas` and plot a heatmap using `seaborn.heatmap`:

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Find correlations:
corr = senic.drop('length', axis=1).corr()

fig = plt.figure(figsize = (10,8))
ax1 = fig.add_subplot(1, 1, 1)

sns.heatmap(corr)

plt.show()
```

b) Combine the predictors into new variables to improve the situation. In particular, substitute `beds` by `pat/beds` and `nurs` by `pat/nurs`.

c) (**Optional, for the advanced reader**)

By using *partial residual plots* you can check whether all predictors have been included in the correct form. How do you interpret partial residual plots?

By analysing the residuals by means of the partial residual plots, explain why the variables `length`, `pat` and `pat.nurs` need to be transformed. Use a log-transformation of these variables.

d) Fit a linear regression model using the log-transformed variables `length`, `pat` and `pat.nurs`.

e) Perform a backward stepwise selection using the AIC criterion. In addition to the already known definitions for `add_one()` and `fit_linear_reg()`, we now need a definition `drop_one`:

[ ]:
```python
def drop_one(x, y, scoreby='RSS'):
    ''' Remove possible predictors from x,
    Fit a linear model on y using fit_linear_reg
    Returns Dataframe showing scores as well as predictor
    to drop in order to keep the best model '''
    # Predefine DataFrame
    x_labels = x.columns
    zeros = np.zeros(len(x_labels))
    results = pd.DataFrame(
        data={'Predictor': x_labels.values, 'BIC': zeros,
              'AIC': zeros, 'R2': zeros,
              'R2_adj': zeros, 'RSS': zeros})

    # For every predictor find RSS and R^2
    for i in range(len(x_labels)):
        x_i = x.drop(columns=x_labels[i])
        results.iloc[i, 1:] = fit_linear_reg(x_i, y)

    # Depending on where we scoreby, we select the highest or lowest
    if scoreby in ['RSS', 'AIC', 'BIC']:
        worst = x_labels[results[scoreby].argmin()]
    elif scoreby in ['R2', 'R2_adj']:
        worst = x_labels[results[scoreby].argmax()]

    return results, worst
```

[ ]:
```python
""" Python Hints: """
results = pd.DataFrame(data={'Worst_Pred': [], 'BIC':[]})

# Define the full predictor
```

```
x_back = x1.copy()

for i in range(x1.shape[1]):
    # Find results and worst predictor using drop_one()

    # Update the full predictor by dropping the worst with pd.drop()

    # Save results

print('Worst Predictors and corresponding BIC:\n', results,
      '\n\nThe best model thus contains',
      x1.shape[1] - results['BIC'].argmin(), ' predictors')
```

Check the final model with the usual diagnostic plots.

f) Now perform a forward stepwise selection using the AIC criterion. Thus, start with the empty model. Check also the diagnostics plots and comment on the differences with respect to c) and d).

g) Perform a hybrid stepwise selection and compare the results you have obtained either through the backward and forward stepwise selection.

h) Carry out an ANOVA-test with the models you have obtained through stepwise selection. You can use `sm.stats.anova_lm()` to execute the ANOVA test.

Exercise 4.3

So-called *Funds of Hedge Funds* (`FoHF`), i.e. portfolios of hedge funds, have different investment strategies with specific returns and risk properties. When such a product is evaluated it is important for the investor to choose the investment style that fits his needs. One approach to assess the investment strategy of a `FoHF` as an outsider is to perform a style analysis based on the returns. Using a regression model (also called multi-factor model in the financial industry) one aims to predict the returns of the `FoHF` on the basis of the returns of the so-called subindices of hedge funds (Long Short Equity, Fixed Income Arbitrage, Global Macro, etc.). The estimated parameters are indications for the chosen investment strategy.

Note that not all investment strategies are present due to the construction of `FoHF`s. The file `FoHF.csv` contains the monthly returns of one `FoHF` and the hedge fund subindices of EDHEC from January 1997 until December 2004. The individual predictors are refered to as the following measures:

- `RV`: Relative value
- `CA`: Convertible Arbitrage

- `FIA`: Fixed Income Arbitrage
- `EMN`: Equity Market Neutral
- `ED`: Event Driven Multistrategy
- `DS`: Distressed Securities
- `MA`: Merger Arbitrage
- `LSE`: Long Short Equity
- `GM`: Global Macro
- `EM`: Emerging Markets
- `CTA`: CTA / Managed Futures
- `SS`: Short Selling

Fit the following model:

$$\text{FoHF} \sim RV + CA + FIA + EMN + ED + DS + MA + LSE + GM + EM + CTA + SS$$

a) Look at the output of `summary()`. What do you conclude with respect to the investment strategy of this `FoHF` when you consider the estimated coefficients, the p-values, the global F-test and the multiple R^2?

b) Check whether this model is valid or whether any assumptions are violated. Also test whether there are problems with respect to multicollinearity and whether all predictors have been included into the model in the correct form.

c) If you have solved the previous subproblem correctly, you will have found some issues. Formulate a strategy how those can be fixed in order to obtain a valid and interpretable result. **Hint**: Creating new predictors is not helpful.

d) Perform variable selection using the BIC criterion. Implement the following search strategies, identify the best/final model and compare:

   i) Hybrid stepwise variable selection, starting with the full model.

   ii) Hybrid stepwise variable selection, starting with the empty model.

e) Does the ANOVA-test justifies that variables have been omitted as a consequence of the stepwise model selections?

f) **Optional**: For this dataset the Lasso is well suited. Fit the model and generate the Lasso traces which allow to identify important predictors. Choose an appropriate value for λ and retrieve the final model as well as its coefficients. See Chapter 6.6 of *Introduction to Statistical Learning* by Gareth et all.

Result Checker

Predictive Modeling

Solutions to Series 4

Solution 4.1 For completeness here the full set of definitions is given:

[ ]:
```
""" Definitions writen and used in Chapter Linear Model Selection """
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt

import seaborn as sns

import statsmodels.api as sm
from statsmodels.graphics.gofplots import ProbPlot
from statsmodels.stats.outliers_influence import variance_inflation_factor


def fit_linear_reg(x, y):
    '''Fit Linear model with predictors x on y
    return AIC, BIC, R2 and R2 adjusted '''
    x = sm.add_constant(x)
    # Create and fit model
    model_k = sm.OLS(y, x).fit()

    # Find scores
    BIC = model_k.bic
    AIC = model_k.aic
    R2 = model_k.rsquared
    R2_adj = model_k.rsquared_adj
    RSS = model_k.ssr

    # Return result in Series
    results = pd.Series(data={'BIC': BIC, 'AIC': AIC, 'R2': R2,
                              'R2_adj': R2_adj, 'RSS': RSS})

    return results


def add_one(x_full, x, y, scoreby='RSS'):
    ''' Add possible predictors from x_full to x,
    Fit a linear model on y using fit_linear_reg
    Returns Dataframe showing scores as well as best model '''
    # Predefine DataFrame
    x_labels = x_full.columns
    zeros = np.zeros(len(x_labels))
    results = pd.DataFrame(
        data={'Predictor': x_labels.values, 'BIC': zeros,
              'AIC': zeros, 'R2': zeros,
              'R2_adj': zeros, 'RSS': zeros})

    # For every predictor find R^2, RSS, and AIC
    for i in range(len(x_labels)):
        x_i = np.concatenate((x, [np.array(x_full[x_labels[i]])]))
        results.iloc[i, 1:] = fit_linear_reg(x_i.T, y)

    # Depending on where we scoreby, we select the highest or lowest
    if scoreby in ['RSS', 'AIC', 'BIC']:
        best = x_labels[results[scoreby].argmin()]
    elif scoreby in ['R2', 'R2_adj']:
        best = x_labels[results[scoreby].argmax()]

    return results, best

def drop_one(x, y, scoreby='RSS'):
    ''' Remove possible predictors from x,
    Fit a linear model on y using fit_linear_reg
    Returns Dataframe showing scores as well as predictor
    to drop in order to keep the best model '''
    # Predefine DataFrame
    x_labels = x.columns
    zeros = np.zeros(len(x_labels))
    results = pd.DataFrame(
        data={'Predictor': x_labels.values, 'BIC': zeros,
              'AIC': zeros, 'R2': zeros,
              'R2_adj': zeros, 'RSS': zeros})

    # For every predictor find RSS and R^2
    for i in range(len(x_labels)):
        x_i = x.drop(columns=x_labels[i])
        results.iloc[i, 1:] = fit_linear_reg(x_i, y)

    # Depending on where we scoreby, we select the highest or lowest
    if scoreby in ['RSS', 'AIC', 'BIC']:
        worst = x_labels[results[scoreby].argmin()]
    elif scoreby in ['R2', 'R2_adj']:
        worst = x_labels[results[scoreby].argmax()]

    return results, worst


""" Plot Residuals vs Fitted Values """
def plot_residuals(axes, res, yfit, n_samp=0):
    """ Inputs:
    axes: axes created with matplotlib.pyplot
    x: x values
    ytrue: y values
    yfit: fitted/predicted y values
    res[optional]: Residuals, used for resampling
    n_samp[optional]: number of resamples """
    # For every random resampling
    for i in range(n_samp):
        # 1. resample indices from Residuals
        samp_res_id = random.sample(list(res), len(res))
        # 2. Average of Residuals, smoothed using LOWESS
        sns.regplot(x=yfit, y=samp_res_id,
                    scatter=False, ci=False, lowess=True,
                    line_kws={'color': 'lightgrey', 'lw': 1, 'alpha': 0.8})
        # 3. Repeat again for n_samples

    dataframe = pd.concat([yfit, res], axis=1)
    axes = sns.residplot(x=yfit, y=res, data=dataframe,
                    lowess=True, scatter_kws={'alpha': 0.5},
                    line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8})
    axes.set_title('Residuals vs Fitted')
    axes.set_ylabel('Residuals')
    axes.set_xlabel('Fitted Values')


""" QQ Plot standardized residuals """
def plot_QQ(axes, res_standard, n_samp=0):
```

```python
    """ Inputs:
    axes: axes created with matplotlib.pyplot
    res_standard: standardized residuals
    n_samp[optional]: number of resamples """
    # QQ plot instance
    QQ = ProbPlot(res_standard)
    # Split the QQ instance in the seperate data
    qqx = pd.Series(sorted(QQ.theoretical_quantiles), name="x")
    qqy = pd.Series(QQ.sorted_data, name="y")
    if n_samp != 0:
        # Estimate the mean and standard deviation
        mu = np.mean(qqy)
        sigma = np.std(qqy)
        # For ever random resampling
        for lp in range(n_samp):
            # Resample indices
            samp_res_id = np.random.normal(mu, sigma, len(qqx))
            # Plot
            sns.regplot(x=qqx, y=sorted(samp_res_id),
                        scatter=False, ci=False, lowess=True,
                        line_kws={'color': 'lightgrey', 'lw': 1, 'alpha': 0.8})

    sns.regplot(x=qqx, y=qqy, scatter=True, lowess=False, ci=False,
                scatter_kws={'s': 40, 'alpha': 0.5},
                line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8})
    axes.plot(qqx, qqx, '--k', alpha=0.5)
    axes.set_title('Normal Q-Q')
    axes.set_xlabel('Theoretical Quantiles')
    axes.set_ylabel('Standardized Residuals')


""" Scale-Location Plot """
def plot_scale_loc(axes, yfit, res_stand_sqrt, n_samp=0):
    """ Inputs:
    axes: axes created with matplotlib.pyplot
    yfit: fitted/predicted y values
    res_stand_sqrt: Absolute square root Residuals
    n_samp[optional]: number of resamples """
    # For every random resampling
    for i in range(n_samp):
        # 1. resample indices from sqrt Residuals
        samp_res_id = random.sample(list(res_stand_sqrt), len(res_stand_sqrt))
        # 2. Average of Residuals, smoothed using LOWESS
        sns.regplot(x=yfit, y=samp_res_id,
                    scatter=False, ci=False, lowess=True,
                    line_kws={'color': 'lightgrey', 'lw': 1, 'alpha': 0.8})
        # 3. Repeat again for n_samples

    # plot Regression usung Seaborn
    sns.regplot(x=yfit, y=res_stand_sqrt,
                scatter=True, ci=False, lowess=True,
                scatter_kws={'s': 40, 'alpha': 0.5},
                line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8})
    axes.set_title('Scale-Location plot')
    axes.set_xlabel('Fitted values')
    axes.set_ylabel('$\sqrt{\|Standardized\ Residuals\|}$')


""" Cook's distance """
def plot_cooks(axes, res_inf_leverage, res_standard, n_pred=1,
               x_lim=None, y_lim=None, n_levels=4):
    """ Inputs:
    axes: axes created with matplotlib.pyplot
    res_inf_leverage: Leverage
    res_standard: standardized residuals
    n_pred: number of predictor variables in x
    x_lim, y_lim[optional]: axis limits
    n_levels: number of levels"""
    sns.regplot(x=res_inf_leverage, y=res_standard,
                scatter=True, ci=False, lowess=True,
                scatter_kws={'s': 40, 'alpha': 0.5},
                line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8})
    # Set limits
    if x_lim != None:
        x_min, x_max = x_lim[0], x_lim[1]
    else:
        x_min, x_max = min(res_inf_leverage)*0.95, max(res_inf_leverage)*1.05
    if y_lim != None:
        y_min, y_max = y_lim[0], y_lim[1]
    else:
        y_min, y_max = min(res_standard)*0.95, max(res_standard)*1.05

    # Plot centre line
    plt.plot((x_min, x_max), (0, 0), 'g--', alpha=0.8)
    # Plot contour lines for Cook's Distance levels
    n = 100
    cooks_distance = np.zeros((n, n))
    x_cooks = np.linspace(x_min, x_max, n)
    y_cooks = np.linspace(y_min, y_max, n)

    for xi in range(n):
        for yi in range(n):
            cooks_distance[yi][xi] = \
            y_cooks[yi]**2 * x_cooks[xi] / (1 - x_cooks[xi]) / (n_pred + 1)
    CS = axes.contour(x_cooks, y_cooks, cooks_distance, levels=n_levels, alpha=0.6)

    axes.clabel(CS, inline=0,  fontsize=10)
    axes.set_xlim(x_min, x_max)
    axes.set_ylim(y_min, y_max)
    axes.set_title('Residuals vs Leverage and Cook\'s distance')
    axes.set_xlabel('Leverage')
    axes.set_ylabel('Standardized Residuals')


""" Standard scatter plot and regression line """
def plot_reg(axes, x, y, x_lab="x", y_lab="y", title="Linear Regression"):
    """ Inputs:
    axes: axes created with matplotlib.pyplot
    x: (single) Feature
    y: Result """
    # Plot scatter data
    sns.regplot(x=x, y=y,
                scatter=True, ci=False, lowess=False,
                scatter_kws={'s': 40, 'alpha': 0.5},
                line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8})
    # Set labels:
    axes.set_xlabel(x_lab)
    axes.set_ylabel(y_lab)
    axes.set_title(title)


""" VIF Analysis """
def VIF_analysis(x):
    """ VIF analysis of variables saved in x
    Input:
    x: m*n matrix or Dataframe, containing m samples of n predictors
    Output:
    VIF: Vector containing n Variance Inflation Factors
    """
    # Preproces:
    x_np = x.to_numpy()
    VIF = []
    # For all n Predictors:
    for i in range(x.shape[1]):
        # Define x and y
        x_i = np.delete(x_np, i, 1)
        x_i = sm.add_constant(x_i)
        y_i = x_np[:, i]
        # Fit model
        model = sm.OLS(y_i, x_i).fit()
        # Calculate the VIF
        VIF.append(1 / (1 - model.rsquared))
    return VIF


""" Residuals Analysis """
def plot_resid_analysis(model):
    # Find the predicted values for the original design.
    yfit = model.fittedvalues
    # Find the Residuals
    res = model.resid
    # Influence of the Residuals
    res_inf = model.get_influence()
    # Studentized residuals using variance from OLS
    res_standard = res_inf.resid_studentized_internal
    # Absolute square root Residuals:
    res_stand_sqrt = np.sqrt(np.abs(res_standard))
    # Cook's Distance and leverage:
    res_inf_cooks = res_inf.cooks_distance
    res_inf_leverage = res_inf.hat_matrix_diag


    """ Plots """
    # Create Figure and subplots
    fig = plt.figure(figsize = (12,9))

    # First subplot: Residuals vs Fitted values with 100 resamples
    ax1 = fig.add_subplot(2, 2, 1)
    plot_residuals(ax1, res, yfit, n_samp = 100)

    # Second subplot: QQ Plot with 100 resamples
    ax2 = fig.add_subplot(2, 2, 2)
    plot_QQ(ax2, res_standard, n_samp = 100)

    # Third subplot: Scale-location with 100 resamples
    ax3 = fig.add_subplot(2, 2, 3)
    plot_scale_loc(ax3, yfit, res_stand_sqrt, n_samp = 100)

    # Fourth subplot: Residuals vs Fitted values with 100 resamples
    ax4 = fig.add_subplot(2, 2, 4)
    plot_cooks(ax4, res_inf_leverage, res_standard, n_pred = model.df_model)

    return
```

a) Python code:

[1]:
```
import pandas as pd

# Load data
fitness = pd.read_csv('./data/fitness.csv')

# As a first inspection, print the first rows of the data:
print(fitness.head())
# As well as the dimensions of the set:
print('\nSize of fitness =\n', fitness.shape)
```

```
   age  weight     oxy  runtime  rstpulse  runpulse  maxpulse
0   44   89.47  44.609    11.37        62       178       182
1   40   75.07  45.313    10.07        62       185       185
2   44   85.84  54.297     8.65        45       156       168
3   42   68.15  59.571     8.17        40       166       172
4   38   89.02  49.874     9.22        55       178       180

Size of fitness =
 (31, 7)
```

[2]:
```
import statsmodels.api as sm
import numpy as np

# Define x and y:
x = fitness.drop('oxy', axis=1)
y = fitness['oxy']

x_sm = sm.add_constant(x)

# Fit the linear model
model = sm.OLS(y, x_sm).fit()

# Print summary
print(model.summary())
```

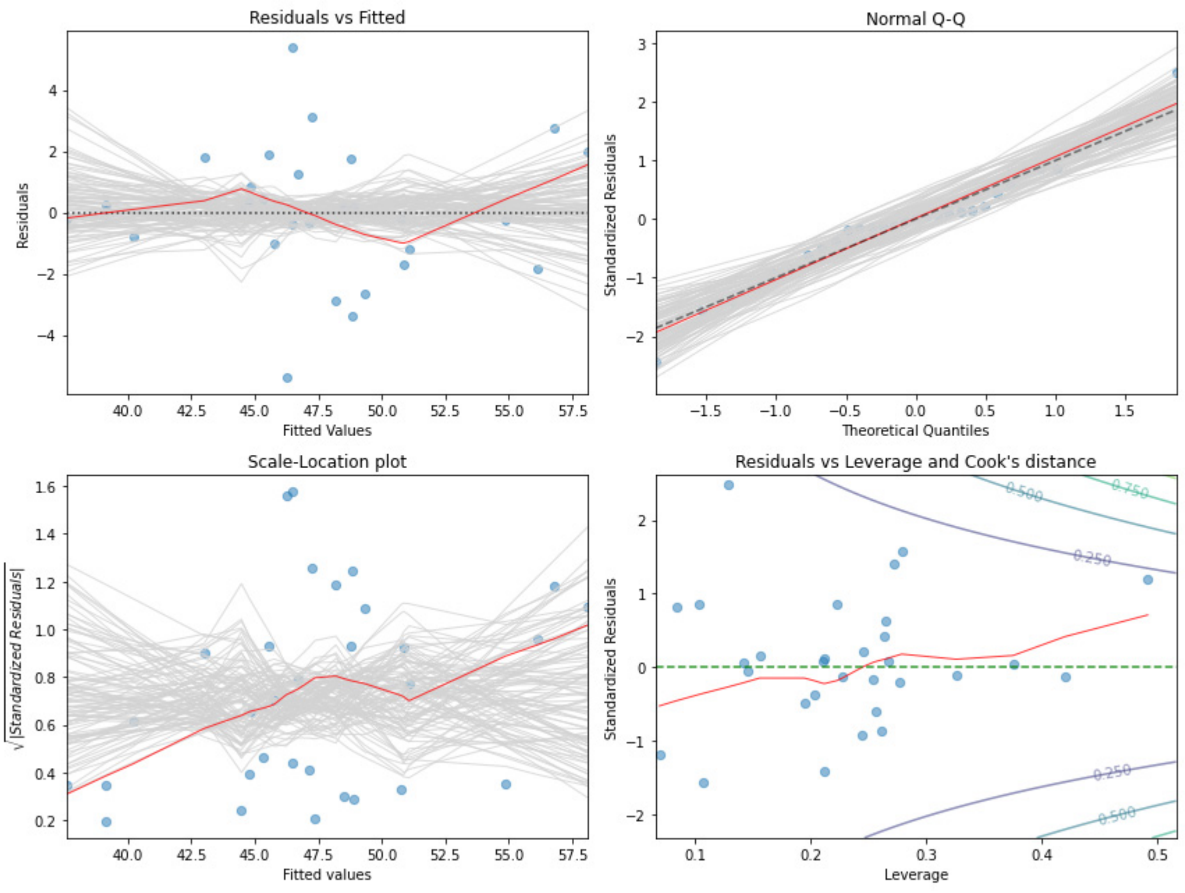
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    oxy   R-squared:                       0.849
Model:                            OLS   Adj. R-squared:                  0.811
Method:                 Least Squares   F-statistic:                     22.43
Date:                Fri, 19 Mar 2021   Prob (F-statistic):           9.72e-09
Time:                        15:10:49   Log-Likelihood:                -66.068
No. Observations:                  31   AIC:                             146.1
Df Residuals:                      24   BIC:                             156.2
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        102.9345     12.403      8.299      0.000      77.335     128.534
age           -0.2270      0.100     -2.273      0.032      -0.433      -0.021
```

```
weight        -0.0742      0.055     -1.359      0.187      -0.187       0.038
runtime       -2.6287      0.385     -6.835      0.000      -3.422      -1.835
rstpulse      -0.0215      0.066     -0.326      0.747      -0.158       0.115
runpulse      -0.3696      0.120     -3.084      0.005      -0.617      -0.122
maxpulse       0.3032      0.136      2.221      0.036       0.022       0.585
==============================================================================
Omnibus:                        2.609   Durbin-Watson:                   1.711
Prob(Omnibus):                  0.271   Jarque-Bera (JB):                1.465
Skew:                          -0.069   Prob(JB):                        0.481
Kurtosis:                       4.056   Cond. No.                     7.91e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 7.91e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Due to the p-value of the associated F-test (**p-value** : $9.715 \cdot 10^{-9}$), we conclude that at least one predictor is associated with the response variable.

b) Note that we use the definitions already mentioned and used in the last question session. **Python** code:

[1]:
```python
from LMS_def import *
import matplotlib.pyplot as plt

def plot_resid_analysis(model):
    # Find the predicted values for the original design.
    yfit = model.fittedvalues
    # Find the Residuals
    res = model.resid
    # Influence of the Residuals
    res_inf = model.get_influence()
    # Studentized residuals using variance from OLS
    res_standard = res_inf.resid_studentized_internal
    # Absolute square root Residuals:
    res_stand_sqrt = np.sqrt(np.abs(res_standard))
    # Cook's Distance and leverage:
    res_inf_cooks = res_inf.cooks_distance
    res_inf_leverage = res_inf.hat_matrix_diag


    """ Plots """
    # Create Figure and subplots
    fig = plt.figure(figsize = (12,9))

    # First subplot: Residuals vs Fitted values with 100 resamples
    ax1 = fig.add_subplot(2, 2, 1)
    plot_residuals(ax1, res, yfit, n_samp = 100)

    # Second subplot: QQ Plot with 100 resamples
    ax2 = fig.add_subplot(2, 2, 2)
    plot_QQ(ax2, res_standard, n_samp = 100)

    # Third subplot: Scale-location with 100 resamples
    ax3 = fig.add_subplot(2, 2, 3)
    plot_scale_loc(ax3, yfit, res_stand_sqrt, n_samp = 100)

    # Fourth subplot: Residuals vs Fitted values with 100 resamples
    ax4 = fig.add_subplot(2, 2, 4)
    plot_cooks(ax4, res_inf_leverage, res_standard, n_pred = model.df_model)

    return

plot_resid_analysis(model)

# Show plot
plt.tight_layout()
plt.show()
```



At first sight, we spot two observations which show relatively large residuals, one of which is positive, the other one is negative. The assumption of constant variance seems to be acceptable at the outermost limit.

Residual Plots (optional, for advanced readers) In many applied problems, it is very interesting to understand and visualize the relation between the response Y and some arbitrary predictor X_k. However, a plot of Y versus X_k probably is deceiving, because in a multiple regression setting, all other predictors $X_1, X_2, \ldots, X_{k-1}, X_{k+1}, \ldots, X_p$ will simultaneously have an effect on the response.

Hence, what we should aim for is displaying the relation of Y versus X_k in the presence of the other predictors. That is what the partial residual plot does. We thus generate an updated Y variable, where the effect of all other predictors is removed from the response.

Mathematically, the partial residuals for predictor X_k are:

$$\begin{aligned} Y - \sum_{j \neq k} \hat{\beta}_k X_k &= \hat{Y} + R - \sum_{j \neq k} \hat{\beta}_k X_k \\ &= \hat{\beta}_k X_k + R \end{aligned}$$

where R denotes the residuals determined for the multiple linear regression model. The residual plots are generated by plotting the partial residuals, that is for every observation i, we plot $\hat{\beta}_k X_i^{(k)} + R_i$ versus the predictor $X_i^{(k)}$.

Although the residual plots do not look perfect, the model assumptions seem to be fulfilled to some limited degree of satisfaction. On the basis of the residual plots we thus conclude that there are no systematic errors.

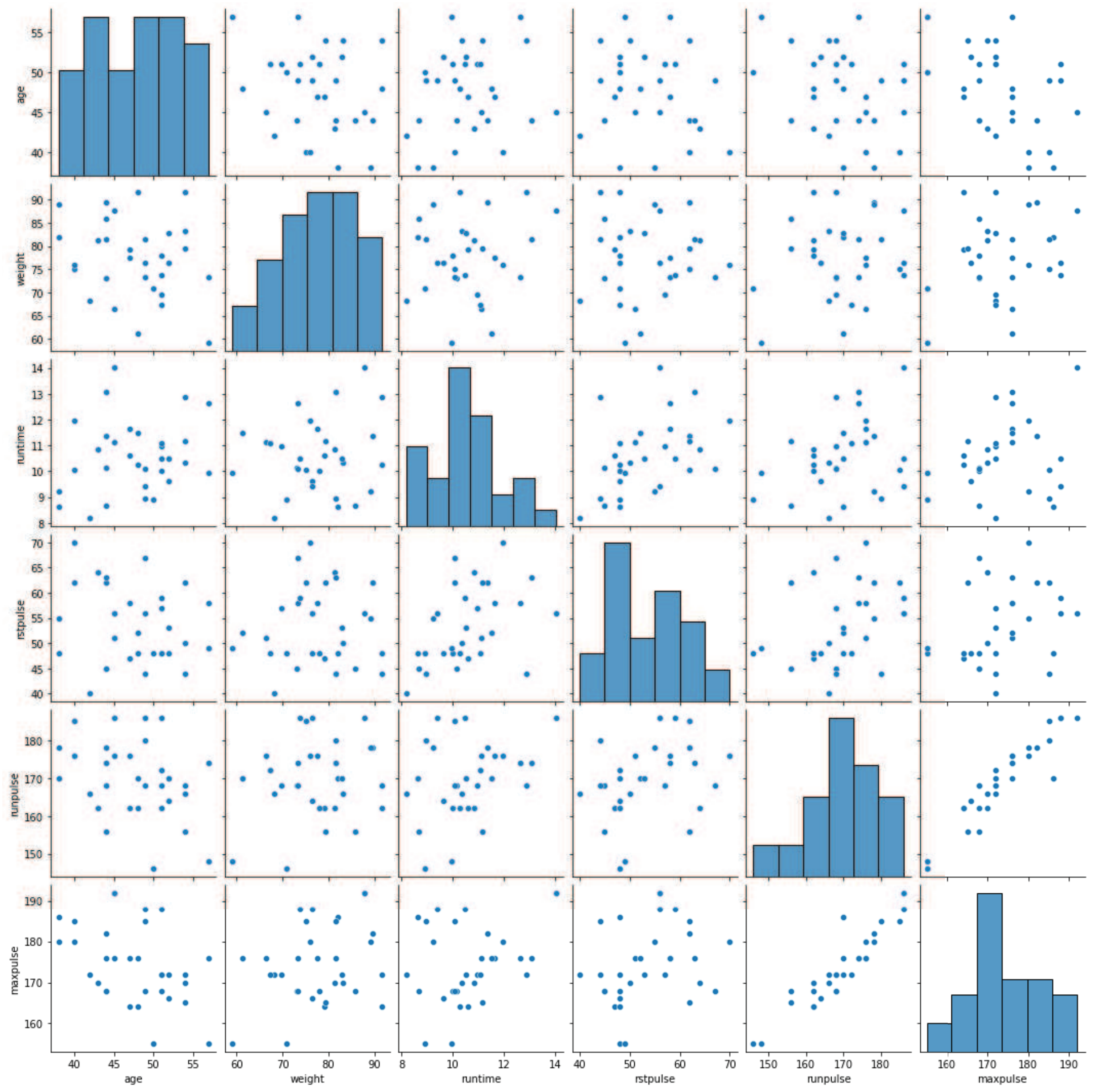
The two observations associated with the large residuals are responsible for deviations in the partial residual plots. In these cases, however, we would not diagnose the presence of a systematic deviation. Therefore, we conclude that the predictors have been included in the model in the correct form.

c) `Python` code:

```
[3]: import seaborn as sns

     fig = sns.pairplot(x)
     plt.show()
```

The analysis of the pairwise correlations should be done without the response variable. As we can see from the above plot, there is a strong positive correlation between the `running pulse` and the `maximal pulse`. The remaining variables do not show strong pairwise correlations.

d) `Python` code:



[5]:
```python
""" Hints: """
import numpy as np
import statsmodels.api as sm
from statsmodels.stats.outliers_influence \
import variance_inflation_factor

# Unfortunately the Method variance_inflation_factor() contains
# an error Therefore we make the analysis ourselves:
def VIF_analysis(x):
    """ VIF analysis of variables saved in x
    Input:
    x: m*n matrix or Dataframe, containing m samples of n predictors
    Output:
```

```
    VIF: Vector containing n Variance Inflation Factors
    """
    # Preproces:
    x_np = x.to_numpy()
    VIF = []
    # For all n Predictors:
    for i in range(x.shape[1]):
        # Define x and y
        x_i = np.delete(x_np, i, 1)
        x_i = sm.add_constant(x_i)
        y_i = x_np[:, i]
        # Fit model
        model = sm.OLS(y_i, x_i).fit()
        # Calculate the VIF
        VIF.append(1 / (1 - model.rsquared))
    return VIF
```

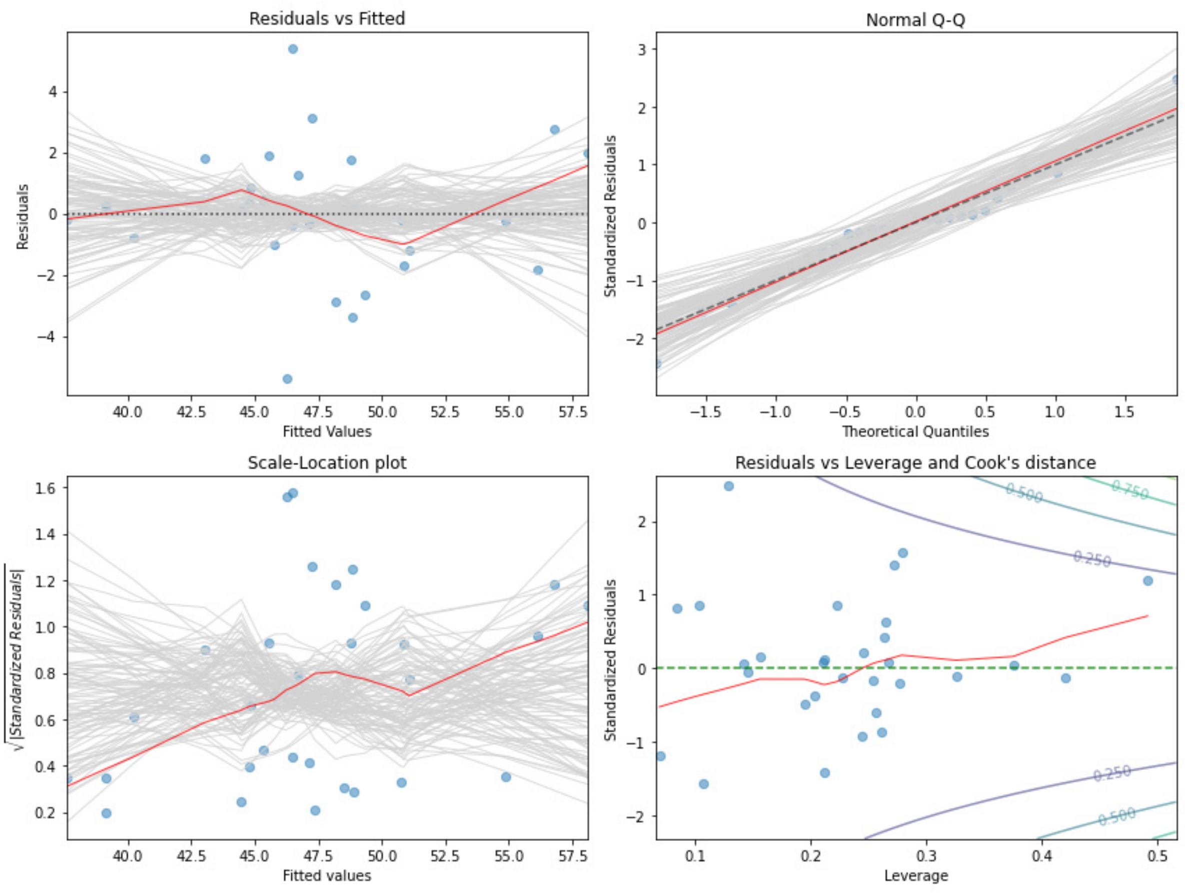
[6]:
```
# VIF analysis:
VIF = VIF_analysis(x)

res = pd.DataFrame(data={'Predictor': x.columns,
                         'VIF': np.round(VIF, 3)})
print(res.T)
```

```
               0       1        2         3         4         5
Predictor    age  weight  runtime  rstpulse  runpulse  maxpulse
VIF        1.513   1.155    1.591     1.416     8.437     8.744
```

The VIFs of **runpulse** and **maxpulse** indicate the presence of critical multicollinearity. This is not surprising given the large pairwise correlation between **running pulse** and **maximal pulse**.

e) i) Amputation **Python** code:

[7]:
```
# Define x and y:
x_i = fitness.drop(['oxy', 'maxpulse'], axis=1)
y = fitness['oxy']

x_sm = sm.add_constant(x_i)

# Fit the linear model
model_i = sm.OLS(y, x_sm).fit()

# Plot using plot function:
plot_resid_analysis(model)

# Show plot
plt.tight_layout()
plt.show()
```



Since the high multicollinearity stems from the large pairwise correlation between `runpulse` and `maxpulse`, one of these two variables should be excluded from the model. We recommend to leave out `maxpulse` due to background knowledge.

Python code:

[8]:
```
# VIF analysis:
VIF = VIF_analysis(x)

res = pd.DataFrame(data={'Predictor': x.columns,
                         'VIF': np.round(VIF, 3)})
print(res.T)
```

```
               0       1        2         3         4
Predictor    age  weight  runtime  rstpulse  runpulse
VIF        1.408   1.116    1.579     1.414     1.389
```

The resulting model does not show any systematic error, the predictors seem to enter the model in the correct form and there is no high multicollinearity.

ii) Either **runpulse** or **maxpulse** need to be adjusted. We keep **runpulse** in the model and substitute **maxpulse** by creating a new variable, which is either **maxpulse - runpulse** or **runpulse/maxpulse**. We will choose as new variable the quotient **runpulse/maxpulse**. **Python** code:

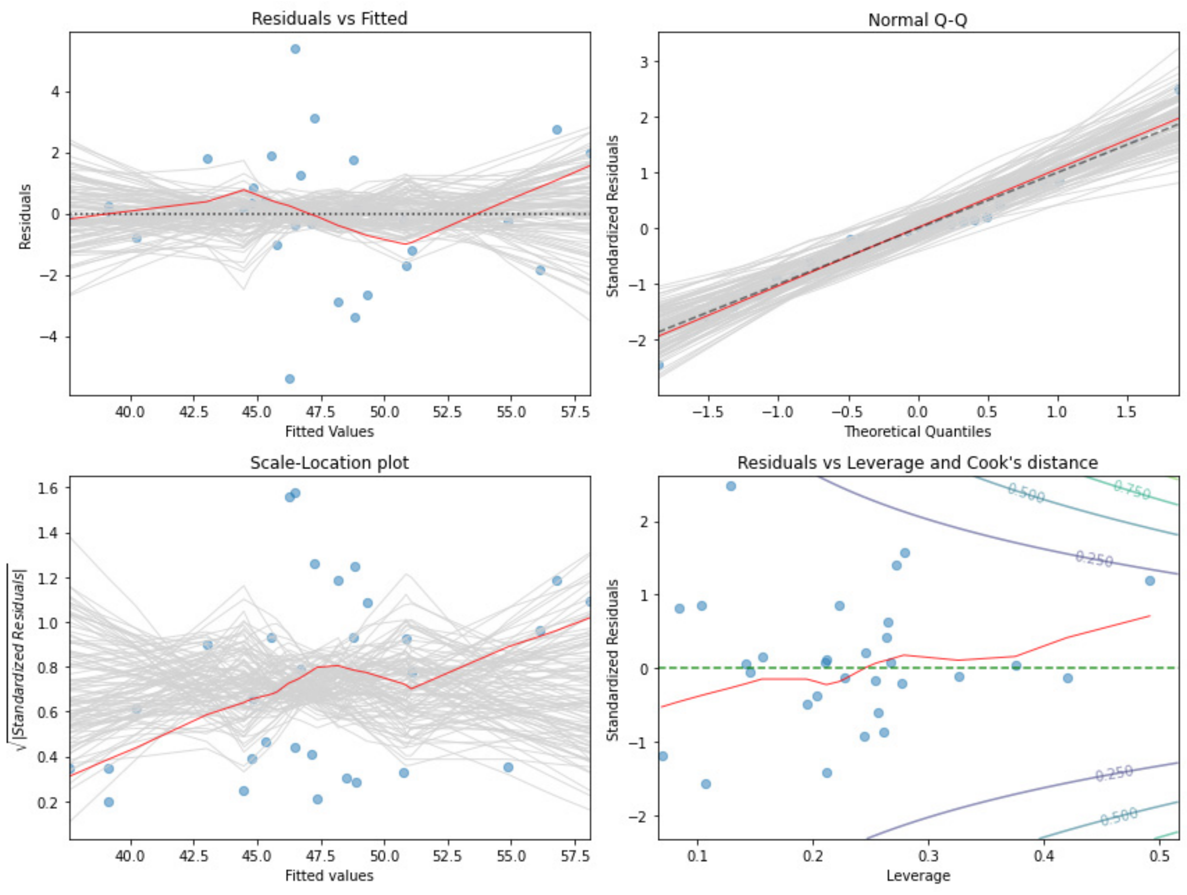
[9]:
```
# Define x and y:
x_ii = fitness.drop(['oxy', 'maxpulse'], axis=1)
newvar = fitness['runpulse'] / fitness['maxpulse']
newvar = newvar.rename('run/max')
x_ii = x_ii.join(newvar)
y = fitness['oxy']

x_sm = sm.add_constant(x_ii)

# Fit the linear model
model_ii = sm.OLS(y, x_sm).fit()

# Plot using plot function:
plot_resid_analysis(model)

# Show plot
plt.tight_layout()
plt.show()
```

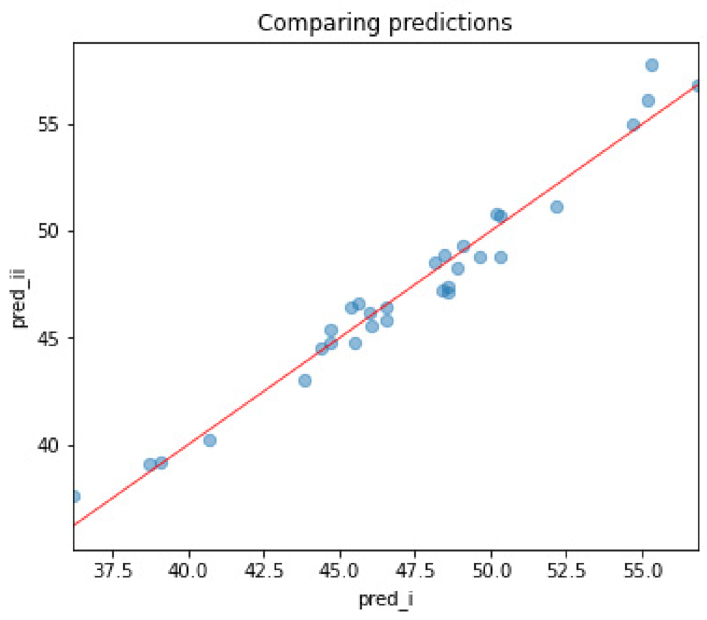
[10]:
```
# VIF analysis:
VIF = VIF_analysis(x_ii)

res = pd.DataFrame(data={'Predictor': x_ii.columns,
                         'VIF': np.round(VIF, 3)})
print(res.T)
```

```
               0       1        2         3         4        5
Predictor    age  weight  runtime  rstpulse  runpulse  run/max
VIF        1.501   1.152    1.594     1.414     1.962    1.616
```

Pairwise comparison of fitted values

[11]:
```
pred_i  = model_i.predict(sm.add_constant(x_i))
pred_ii = model_ii.predict(sm.add_constant(x_ii))


fig = plt.figure(figsize = (6,5))
ax1 = fig.add_subplot(1, 1, 1)

plot_reg(ax1, pred_i, pred_ii,
         x_lab="pred_i", y_lab="pred_ii",
         title="Comparing predictions")
plt.show()
```





When using amputation the fitted values are notably different from the approach using a new variable. This indicates that we do lose some precision when excluding a variable from the set of predictors.

f) We will use the same definition as made in the Chapter Linear Model Selection:

[12]:
```python
def fit_linear_reg(x, y):
    '''Fit Linear model with predictors x on y
    return AIC, BIC, R2 and R2 adjusted '''
    x = sm.add_constant(x)
    # Create and fit model
    model_k = sm.OLS(y, x).fit()

    # Find scores
    BIC = model_k.bic
    AIC = model_k.aic
    R2 = model_k.rsquared
    R2_adj = model_k.rsquared_adj
    RSS = model_k.ssr

    # Return result in Series
    results = pd.Series(data={'BIC': BIC, 'AIC': AIC, 'R2': R2,
                              'R2_adj': R2_adj, 'RSS': RSS})

    return results


def add_one(x_full, x, y, scoreby='RSS'):
    ''' Add possible predictors from x_full to x,
    Fit a linear model on y using fit_linear_reg
    Returns Dataframe showing scores as well as best model '''
    # Predefine DataFrame
    x_labels = x_full.columns
    zeros = np.zeros(len(x_labels))
    results = pd.DataFrame(
        data={'Predictor': x_labels.values, 'BIC': zeros,
              'AIC': zeros, 'R2': zeros,
              'R2_adj': zeros, 'RSS': zeros})

    # For every predictor find R^2, RSS, and AIC
    for i in range(len(x_labels)):
        x_i = np.concatenate((x, [np.array(x_full[x_labels[i]])]))
        results.iloc[i, 1:] = fit_linear_reg(x_i.T, y)

    # Depending on where we scoreby, we select the highest or lowest
    if scoreby in ['RSS', 'AIC', 'BIC']:
        best = x_labels[results[scoreby].argmin()]
    elif scoreby in ['R2', 'R2_adj']:
        best = x_labels[results[scoreby].argmax()]

    return results, best
```

[13]:
```python
# Define the empty dataframe and the empty predictor
results = pd.DataFrame(data={'Best_Pred': [], 'AIC':[]})

# Define the empty predictor
x0 = [np.zeros(len(y))]

x_red = x_ii.copy() # Copy the full set of predictors
x_forward = x0

for i in range(x_red.shape[1]):
    results_i, best_i = add_one(x_red, x_forward, y, scoreby='AIC')

    # Update the empty predictor with the best predictor
    x_forward = np.concatenate((x_forward, [x_ii[best_i]]))

    # Remove the chosen predictor from the list of options
    x_red = x_red.drop(columns=best_i)

    # Save results
    results.loc[i, 'Best_Pred'] = best_i
    results.loc[i, 'AIC'] = results_i['AIC'].min()

print('Best Predictors and corresponding AIC:\n', results,
      '\n\nThe best model thus contains',
      results['AIC'].argmin() + 1, ' predictors')
```

```
Best Predictors and corresponding AIC:
   Best_Pred         AIC
0    runtime  152.508320
1    run/max  145.859443
2        age  144.720270
3     weight  144.306820
4   runpulse  144.847297
5   rstpulse  146.671835

The best model thus contains 4  predictors
```

The variables **`rstpulse`** and **`runpulse`** are excluded from the model. Finally, only **`runtime`**, **`age`** and **`intensity`** (the quotient of **`runpulse`** and **`maxpulse`**) are considered in the model. **`Python`** code:

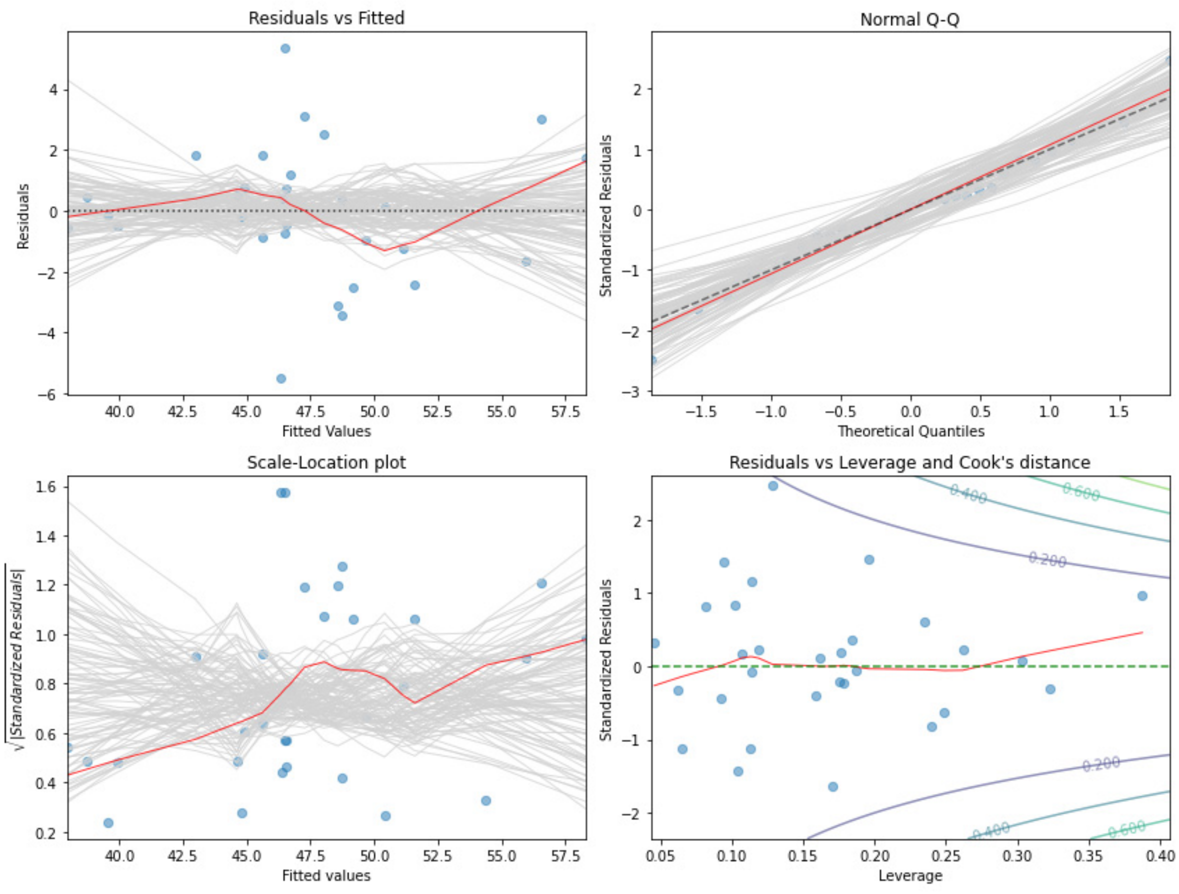
[14]:
```python
# Define the optimal model:
x_opt = x_ii[results.iloc[:4, 0]]

x_sm = sm.add_constant(x_opt)

# Fit the linear model
model_opt = sm.OLS(y, x_sm).fit()
```

```
# Table and print results
table = sm.stats.anova_lm(model_ii, model_opt)
print(table)
```

```
   df_resid         ssr  df_diff   ss_diff         F  Pr(>F)
0      24.0  131.085102      0.0       NaN       NaN     NaN
1      26.0  138.184286     -2.0 -7.099184  0.667872     NaN
```

According to the ANOVA-test, the null hypothesis $\beta_{\text{rstpulse}} = \beta_{\text{runpulse}} = 0$ cannot be rejected. Thus the model selection consequently omitted the corresponding predictor variables. **Python** code:

[15]:
```
# Plot using plot function:
plot_resid_analysis(model_opt)

# Show plot
plt.tight_layout()
plt.show()
```



The residual plots do not point to any systematic errors.

g) According to our results, the rate of oxygen consumption could be modeled with the variables `running time`, `age` and `intensity`. This yields an R^2 value of approximatly 0.82, i.e., the rate of oxygen consumption can be explained rather well, however not perfectly. It is difficult to decide whether this is sufficient for practical purposes. We thus cannot conclude on the basis of our results whether this model is applicable. The trade-off between costs and loss of precision would need to be assessed further.

Solution 4.2 `Python` code:

[1]:
```
import pandas as pd

# Load data
senic = pd.read_csv('./data/senic.csv')

# As a first inspection, print the first rows of the data:
print(senic.head())
# As well as the dimensions of the set:
print('\nSize of senic =\n', senic.shape)
```

```
   id  length   age  inf  cult   xray  beds  school region  pat  nurs  serv
0   1    7.13  55.7  4.1   9.0   39.6   279       2      W  207   241  60.0
1   2    8.82  58.2  1.6   3.8   51.7    80       2      N   51    52  40.0
2   3    8.34  56.9  2.7   8.1   74.0   107       2      S   82    54  20.0
3   4    8.95  53.7  5.6  18.9  122.8   147       2      W   53   148  40.0
4   5   11.20  56.5  5.7  34.5   88.9   180       2     NE  134   151  40.0

Size of senic =
 (113, 12)
```

[2]:
```
# Convert Categorical variables
senic = pd.get_dummies(data=senic, drop_first=True,
                    columns = ['school', 'region'],
                    prefix=('school', 'region'))

# Remove unused variables:
senic.rename(columns={'beds': 'pat/beds', 'nurs': 'pat/nurse'},
             inplace=True)

print(senic.head())
```

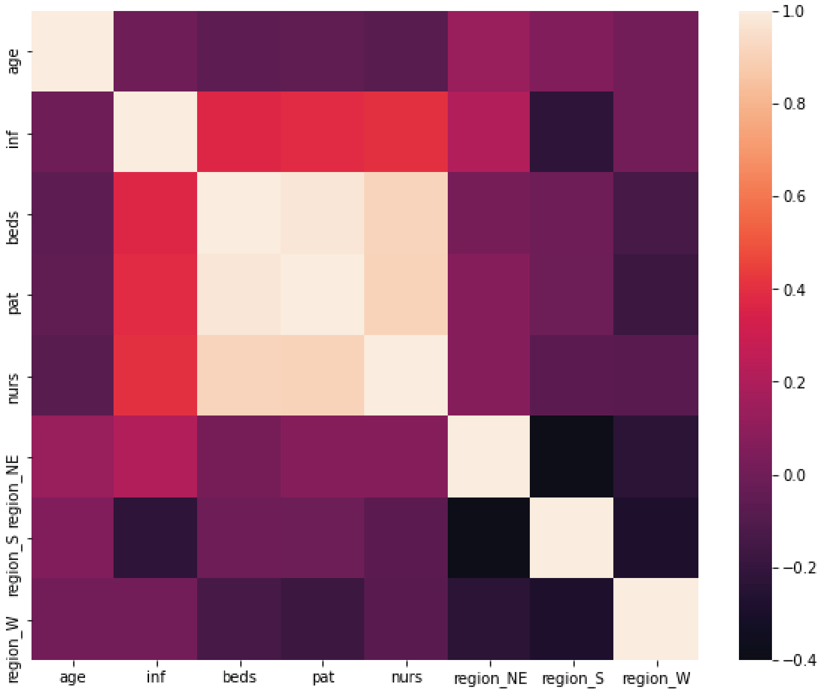
```
   length   age  inf  beds  pat  nurs  region_NE  region_S  region_W
0    7.13  55.7  4.1   279  207   241          0         0         1
1    8.82  58.2  1.6    80   51    52          0         0         0
2    8.34  56.9  2.7   107   82    54          0         1         0
3    8.95  53.7  5.6   147   53   148          0         0         1
4   11.20  56.5  5.7   180  134   151          1         0         0
```

a) We check the correlations between the continuous predictors (without response variable): `Python` code:

[3]:
```
# Find correlations:
corr = senic.drop('length', axis=1).corr()
```

Graphical illustration of correlations: **Python** code:

[1]:
```
import seaborn as sns
import matplotlib.pyplot as plt

fig = plt.figure(figsize = (10,8))
ax1 = fig.add_subplot(1, 1, 1)

sns.heatmap(corr)

plt.show()
```



We observe that **beds**, **pat** and **nurs** are strongly correlated. This is to be expected since these variables all measure the size of a hospital.

b) We will leave the variable **pat** unmodified because it is definitely a key factor to be taken into account when **length** is the response variable. We transform the other variables to solve the high-correlation problem without having to omit them in the regression model. Hence, we will substitute **beds** by **pat**/**beds** and **nurs** by **pat**/**nurs**.

Before combining the variables, we check whether **beds** and **nurs** contain zeros:

```
[5]: print('Zeros in beds:\n', senic['beds'].isin([0]).sum(),
           '\nZeros in nurs:\n', senic['nurs'].isin([0]).sum())
```

```
Zeros in beds:
 0
Zeros in nurs:
 0
```

Now we combine the variables to new variables and check the correlations again. **Python** code:

```
[6]: senic['beds'] = senic['pat'] / senic['beds']
     senic['nurs'] = senic['pat'] / senic['nurs']
     # Rename cols:
     x1.rename(columns={'pat': 'log(pat)', 'pat/nurse': 'log(pat/nurse)'},
               inplace=True)

     print(senic.head())
```

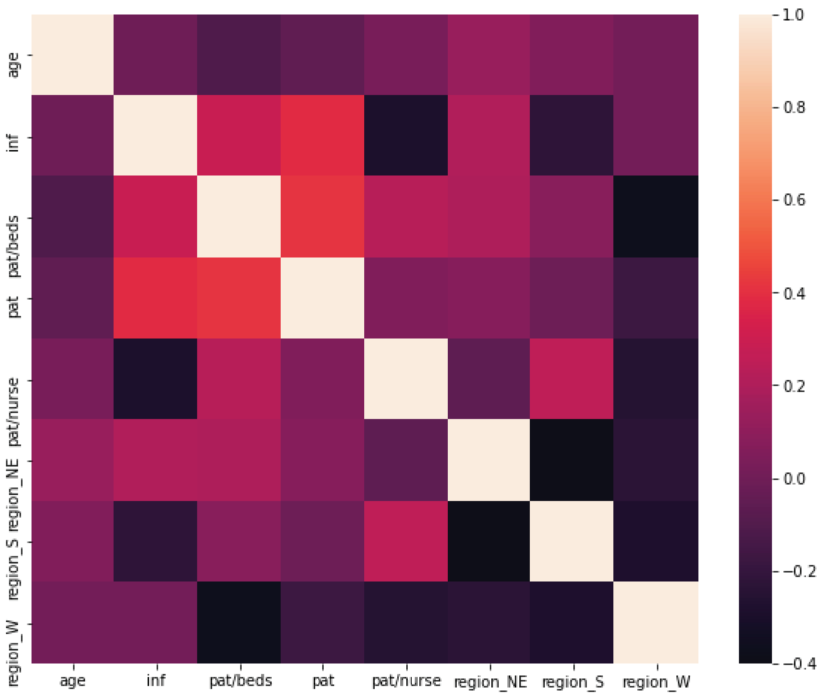
```
   length   age  inf  pat/beds  pat  pat/nurse  region_NE  region_S  region_W
0    7.13  55.7  4.1  0.741935  207   0.858921          0         0         1
1    8.82  58.2  1.6  0.637500   51   0.980769          0         0         0
2    8.34  56.9  2.7  0.766355   82   1.518519          0         1         0
3    8.95  53.7  5.6  0.360544   53   0.358108          0         0         1
4   11.20  56.5  5.7  0.744444  134   0.887417          1         0         0
```

```
[7]: # Find correlations:
     corr = senic.drop('length', axis=1).corr()

     fig = plt.figure(figsize = (10,8))
     ax1 = fig.add_subplot(1, 1, 1)

     sns.heatmap(corr)

     plt.show()
```

The correlations could be strongly reduced and we still dispose of information with respect to the variables **beds** and **nurs**.

c) **Residual Plots (optional, for advanced readers)** In many applied problems, it is very interesting to understand and visualize the relation between the response Y and some arbitrary predictor X_k. However, a plot of Y versus X_k probably is deceiving, because in a multiple regression setting, all other predictors $X_1, X_2, \dots, X_{k-1}, X_{k+1}, \dots, X_p$ will simultaneously have an effect on the response.

Hence, what we should aim for is displaying the relation of Y versus X_k in the presence of the other predictors. That is what the partial residual plot does. We



thus generate an updated Y variable, where the effect of all other predictors is removed from the response.

Mathematically, the partial residuals for predictor X_k are:

$$\begin{aligned} Y - \sum_{j \neq k} \hat{\beta}_k X_k &= \hat{Y} + R - \sum_{j \neq k} \hat{\beta}_k X_k \\ &= \hat{\beta}_k X_k + R \end{aligned}$$

where R denotes the residuals determined for the multiple linear regression model. The residual plots are generated by plotting the partial residuals, that is for every observation i, we plot $\hat{\beta}_k X_i^{(k)} + R_i$ versus the predictor $X_i^{(k)}$.

d) We fit a linear regression model: **Python** code:

[8]:
```python
import statsmodels.api as sm
import numpy as np

# Define x and y:
x = senic.drop('length', axis=1)
x1 = x.copy()
y = np.log(senic['length'])

# log transform x:
x1[['pat', 'pat/nurse']] = np.log(x1[['pat', 'pat/nurse']])
```

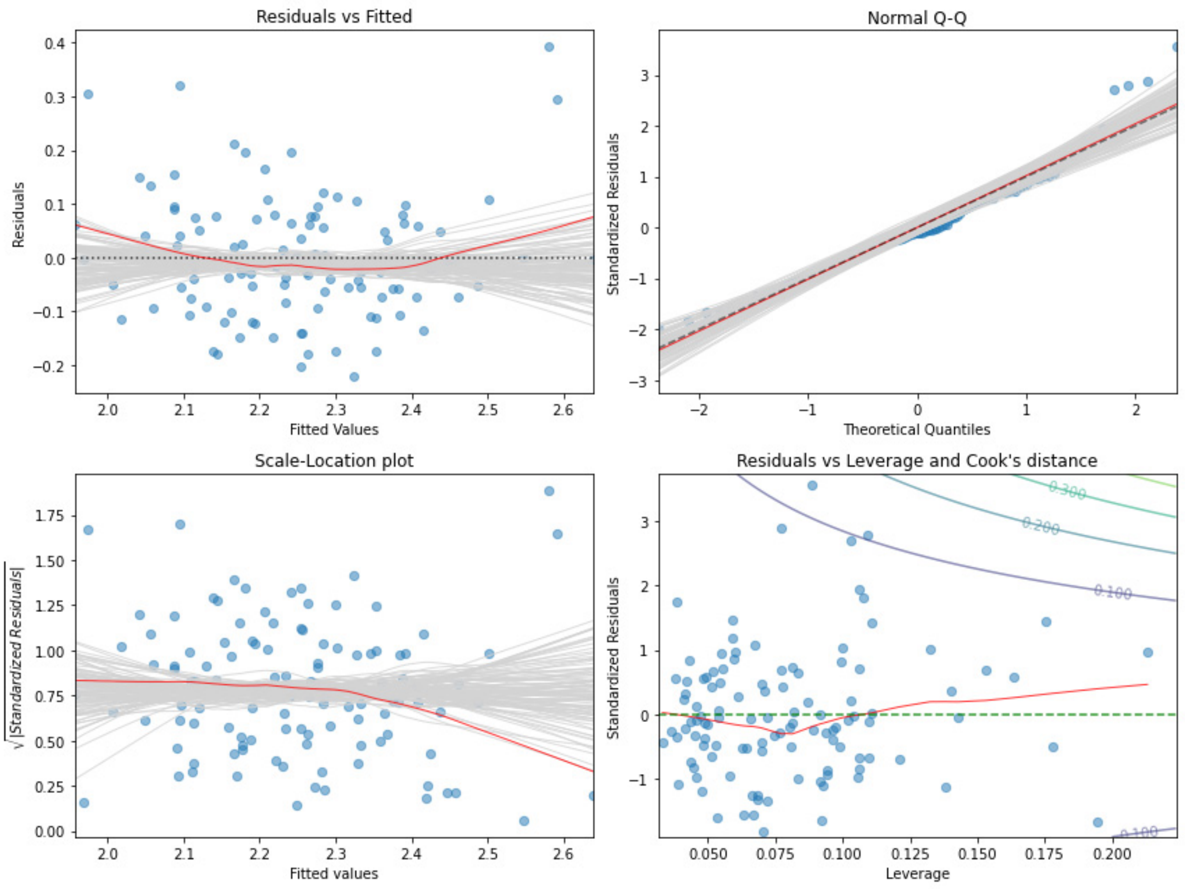
```
x1.rename(columns={'pat': 'log(pat)', 'pat/nurse': 'log(pat/nurse)'},
          inplace=True)

x_sm = sm.add_constant(x1)

# Fit the linear model
model_1 = sm.OLS(y, x_sm).fit()

# Print summary
print(model_1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 length   R-squared:                       0.608
Model:                            OLS   Adj. R-squared:                  0.578
Method:                 Least Squares   F-statistic:                     20.17
Date:                Fri, 19 Mar 2021   Prob (F-statistic):           4.28e-18
Time:                        18:01:47   Log-Likelihood:                 88.141
No. Observations:                 113   AIC:                            -158.3
Df Residuals:                     104   BIC:                            -133.7
Df Model:                           8
Covariance Type:            nonrobust
==================================================================================
                     coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------------
const              1.3055      0.171      7.617      0.000       0.966       1.645
age                0.0076      0.003      2.997      0.003       0.003       0.013
inf                0.0539      0.010      5.228      0.000       0.033       0.074
pat/beds           0.1064      0.124      0.856      0.394      -0.140       0.353
log(pat)           0.0470      0.018      2.643      0.009       0.012       0.082
log(pat/nurse)     0.0738      0.037      1.985      0.050    6.3e-05       0.148
region_NE          0.0741      0.031      2.379      0.019       0.012       0.136
region_S          -0.0473      0.029     -1.638      0.105      -0.105       0.010
region_W          -0.1264      0.038     -3.303      0.001      -0.202      -0.050
==============================================================================
Omnibus:                       15.658   Durbin-Watson:                   1.940
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               18.819
Skew:                           0.788   Prob(JB):                     8.19e-05
Kurtosis:                       4.230   Cond. No.                         902.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

From the summary output we conclude that `pat.bed` is statistically not significant and a variable selection may be appropriate (see next question). `Python` code:

```
[1]: # using our set of definitions:
     from LMS_def import *

     # Plot using plot function:
     plot_resid_analysis(model_1)

     # Show plot
```

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



Based on the model diagnostics plots we note that there are three outliers, i.e. observations 47, 101, and 112. However, since their Cook's distance is below 0.5, they do not significantly influence the model fit and we proceed with our analysis. The assumptions concerning linearity and constant variance seem to be satisfied. The QQ-plot does not look perfect but we may assume that the normality assumption still is fulfilled. Now we visualize our model by means of partial residual plots. As it can be deduced from these plots, the predictor **pat.bed** does not have much explanatory power, and indeed, its p-value is rather large as we have seen in the **R** summary.

e) Backward stepwise selection: **Python** code:

[10]:
```
""" Functions as defined in LMS_def: """
def fit_linear_reg(x, y):
    '''Fit Linear model with predictors x on y
    return AIC, BIC, R2 and R2 adjusted '''
    x = sm.add_constant(x)
    # Create and fit model
    model_k = sm.OLS(y, x).fit()

    # Find scores
    BIC = model_k.bic
    AIC = model_k.aic
    R2 = model_k.rsquared
    R2_adj = model_k.rsquared_adj
    RSS = model_k.ssr

    # Return result in Series
    results = pd.Series(data={'BIC': BIC, 'AIC': AIC, 'R2': R2,
                              'R2_adj': R2_adj, 'RSS': RSS})

    return results


def add_one(x_full, x, y, scoreby='RSS'):
    ''' Add possible predictors from x_full to x,
    Fit a linear model on y using fit_linear_reg
    Returns Dataframe showing scores as well as best model '''
    # Predefine DataFrame
    x_labels = x_full.columns
    zeros = np.zeros(len(x_labels))
    results = pd.DataFrame(
        data={'Predictor': x_labels.values, 'BIC': zeros,
               'AIC': zeros, 'R2': zeros,
               'R2_adj': zeros, 'RSS': zeros})

    # For every predictor find R^2, RSS, and AIC
    for i in range(len(x_labels)):
        x_i = np.concatenate((x, [np.array(x_full[x_labels[i]])]))
        results.iloc[i, 1:] = fit_linear_reg(x_i.T, y)

    # Depending on where we scoreby, we select the highest or lowest
    if scoreby in ['RSS', 'AIC', 'BIC']:
        best = x_labels[results[scoreby].argmin()]
    elif scoreby in ['R2', 'R2_adj']:
        best = x_labels[results[scoreby].argmax()]

    return results, best

def drop_one(x, y, scoreby='RSS'):
    ''' Remove possible predictors from x,
    Fit a linear model on y using fit_linear_reg
    Returns Dataframe showing scores as well as predictor
    to drop in order to keep the best model '''
    # Predefine DataFrame
    x_labels = x.columns
    zeros = np.zeros(len(x_labels))
    results = pd.DataFrame(
        data={'Predictor': x_labels.values, 'BIC': zeros,
              'AIC': zeros, 'R2': zeros,
              'R2_adj': zeros, 'RSS': zeros})

    # For every predictor find RSS and R^2
    for i in range(len(x_labels)):
        x_i = x.drop(columns=x_labels[i])
        results.iloc[i, 1:] = fit_linear_reg(x_i, y)

    # Depending on where we scoreby, we select the highest or lowest
    if scoreby in ['RSS', 'AIC', 'BIC']:
        worst = x_labels[results[scoreby].argmin()]
    elif scoreby in ['R2', 'R2_adj']:
        worst = x_labels[results[scoreby].argmax()]

    return results, worst
```

[11]:

```
results = pd.DataFrame(data={'Worst_Pred': [], 'BIC':[]})

# Define the full predictor
x_back = x1.copy()

for i in range(x1.shape[1]):
    results_i, worst_i = drop_one(x_back, y, scoreby='BIC')

    # Update the empty predictor with the best predictor
    x_back = x_back.drop(columns=worst_i)

    # Save results
    results.loc[i, 'Worst_Pred'] = worst_i
    results.loc[i, 'BIC'] = results_i['BIC'].min()

print('Worst Predictors and corresponding BIC:\n', results,
      '\n\nThe best model thus contains',
      x1.shape[1] - results['BIC'].argmin(), ' predictors')
```

```
Worst Predictors and corresponding BIC:
         Worst_Pred         BIC
0         pat/beds -137.670667
1         region_S -139.743552
2   log(pat/nurse) -139.911888
3              age -136.551672
4         log(pat) -131.179926
5        region_NE -123.963321
6         region_W -102.130871
7              inf  -65.703326

The best model thus contains 6  predictors
```

The backward stepwise selection using AIC removes only the variable `pat.bed` from the model, just as the backward stepwise selection using AIC.

We will now save the model for future reference:

[12]:
```
# Define the optimal model:
x_back = x1[results.iloc[:7, 0]]

x_sm = sm.add_constant(x_back)

# Fit the linear model
model_back = sm.OLS(y, x_sm).fit()
```

f) Forward stepwise selection: **Python** code:

[13]:
```
# Define the empty dataframe and the empty predictor
results = pd.DataFrame(data={'Best_Pred': [], 'AIC':[]})

# Define the empty predictor
x0 = [np.zeros(len(y))]

x_red = x1.copy() # Copy the full set of predictors
x_forward = x0

for i in range(x_red.shape[1]):
    results_i, best_i = add_one(x_red, x_forward, y, scoreby='AIC')

    # Update the empty predictor with the best predictorc
    x_forward = np.concatenate((x_forward, [x1[best_i]]))

    # Remove the chosen predictor from the list of options
    x_red = x_red.drop(columns=best_i)

    # Save results
    results.loc[i, 'Best_Pred'] = best_i
    results.loc[i, 'AIC'] = results_i['AIC'].min()

print('Best Predictors and corresponding AIC:\n', results,
      '\n\nThe best model thus contains',
      results['AIC'].argmin() + 1, ' predictors')
```

```
Best Predictors and corresponding AIC:
          Best_Pred         AIC
0              inf -107.585646
1         region_W -132.145485
2        region_NE -142.089477
3         log(pat) -150.188612
4              age -156.276215
5   log(pat/nurse) -158.835267
6         region_S -159.489769
```

```
7        pat/beds -158.282941

The best model thus contains 7  predictors
```

We obtain the same result as for the backward stepwise selection using AIC : only the predictor variable `pat.bed` has been removed from the model. Note that this occurs in this particular example and represents rather an exception.

[1]:
```
# Define the optimal model:
x_forw = x1[results.iloc[:7, 0]]

x_sm = sm.add_constant(x_forw)

# Fit the linear model
model_forw = sm.OLS(y, x_sm).fit()
```

g) Hybrid stepwise selection:

Hybrid stepwise selection starting with the empty model yields the same result as the hybrid stepwise starting with the full model, backward stepwise selection and forward stepwise selection. Note that this is in general not the case: applying these methods with different data could actually lead to different results.

h) The ANOVA-test yields `Python` code:

[15]:
```
# Table and print results
table = sm.stats.anova_lm(model_1, model_back, model_forw)
print(table)
```

```
   df_resid       ssr  df_diff   ss_diff          F  Pr(>F)
0     104.0  1.390252      0.0       NaN        NaN     NaN
1     105.0  1.755690     -1.0 -0.365438  27.406999     NaN
2     105.0  1.400044     -0.0  0.355645       -inf     NaN
```

We conclude that the predictor variable `pat.bed` can indeed be dropped from the model because of the p-value associated with the F-statistic. We further note that the p-value associated with the F-statistic is in this case identical to the p-value associated with the t-statistic which we have observed in `R`-output of the full regression model.

Solution 4.3

a) `Python` code:

[1]:
```
import pandas as pd

# loading data
```

```python
df_FoHF = pd.read_csv('./data/FoHF.csv')

# inspecting data
df_FoHF.head()
df_FoHF.tail()
df_FoHF.dtypes
df_FoHF.describe(include = 'all')


# defining predictors x and response variable y
x = df_FoHF.drop('FoHF', axis = 1)
y = df_FoHF['FoHF']

# defining design matrix (model matrix or regressor matrix) X
X = sm.add_constant(x)

# fitting linear model with all predictors
model_full = sm.OLS(y, X).fit()

# printing model summary
print(model_full.summary())
```

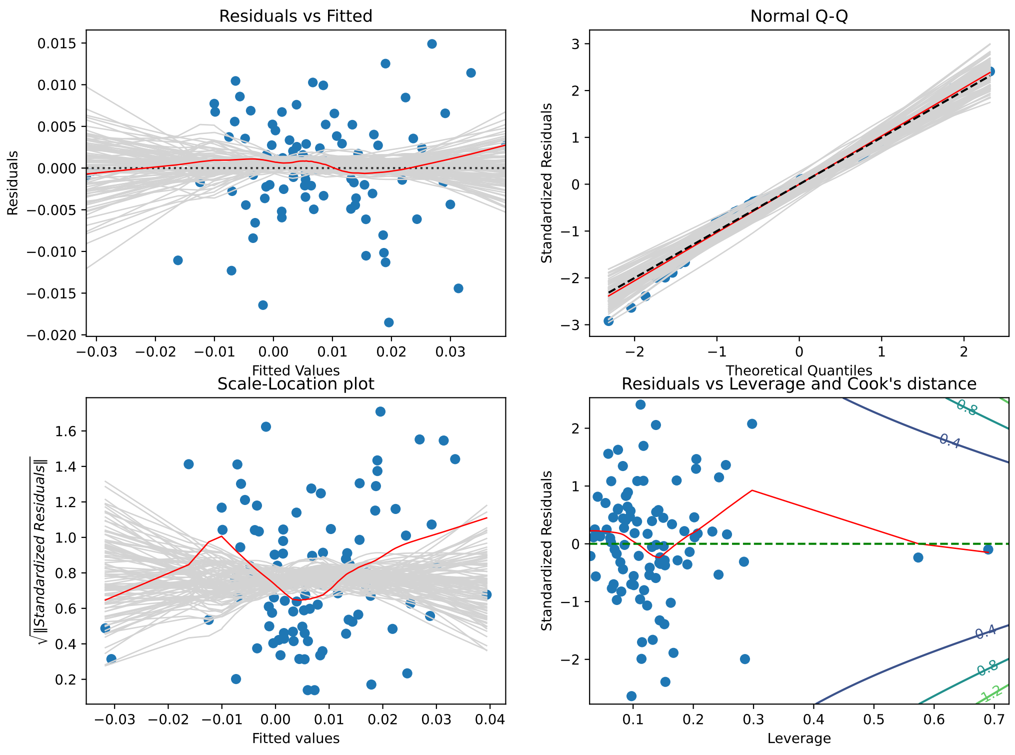
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   FoHF   R-squared:                       0.808
Model:                            OLS   Adj. R-squared:                  0.780
Method:                 Least Squares   F-statistic:                     29.03
Date:                Thu, 05 Jun 2025   Prob (F-statistic):           1.01e-24
Time:                        15:32:12   Log-Likelihood:                 353.29
No. Observations:                  96   AIC:                            -680.6
Df Residuals:                      83   BIC:                            -647.2
Df Model:                          12
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0023      0.001     -1.736      0.086      -0.005       0.000
RV            -0.3889      0.171     -2.272      0.026      -0.729      -0.048
CA             0.2387      0.105      2.283      0.025       0.031       0.447
FIA            0.3630      0.088      4.133      0.000       0.188       0.538
EMN            0.1848      0.197      0.936      0.352      -0.208       0.578
ED             0.3149      0.216      1.459      0.148      -0.114       0.744
DS            -0.0077      0.124     -0.062      0.951      -0.255       0.240
MA            -0.0284      0.169     -0.168      0.867      -0.365       0.309
LSE            0.1536      0.100      1.543      0.127      -0.044       0.352
GM             0.1271      0.087      1.463      0.147      -0.046       0.300
EM             0.0492      0.035      1.403      0.164      -0.021       0.119
CTA            0.1592      0.037      4.268      0.000       0.085       0.233
SS             0.0326      0.023      1.393      0.167      -0.014       0.079
==============================================================================
Omnibus:                        4.964   Durbin-Watson:                   1.520
Prob(Omnibus):                  0.084   Jarque-Bera (JB):                4.378
Skew:                          -0.408   Prob(JB):                        0.112
Kurtosis:                       3.654   Cond. No.                         429.
==============================================================================

Notes:
```

```
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Only four variables are significant at the 5 % level in the summary output; two variables are associated with a very small p-value. The multiple R^2 is relatively large having a value of around 80 %. The global F-test is highly significant. In other words, the return of the FoHF is explained rather well and not all subindices may be necessary to predict the return of the FoHF. Therefore, we can assume that the FoHF does not invest in all subindices.

b) Python code:

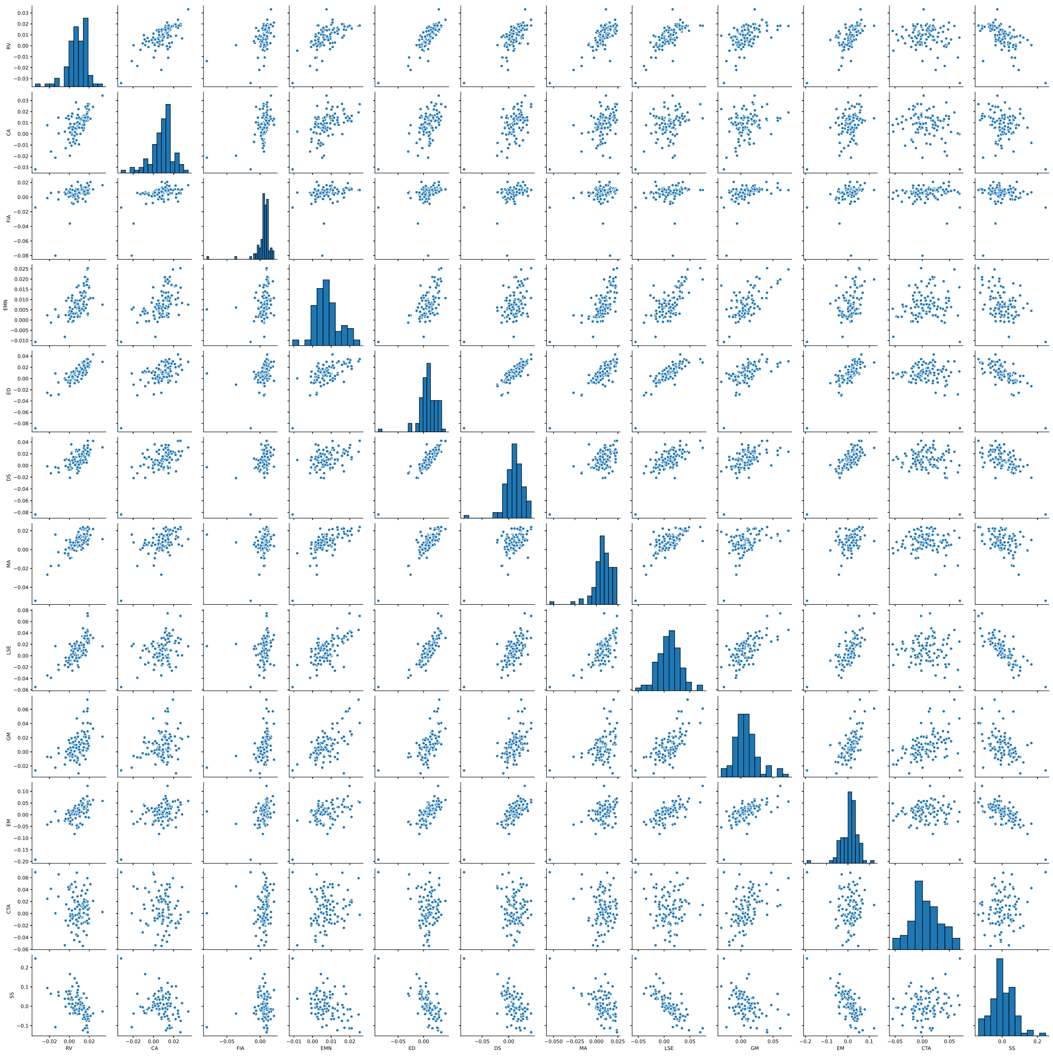
```
[1]: import matplotlib.pyplot as plt
     from LMS_def import *

     plot_resid_analysis(model_full)
```



The residual plots do not point to any systematic errors in the model. The normality assumption seems to be satisfied. The smoother in the scale location plot does show some deviations from the horizontal line. It is generally known that finance data shows volatility, i.e. the (conditional) variance is not necessarily constant over time. However, in this case the deviations are not very pronounced, so that the constant variance assumption seems to be satisfied and we can proceed with the analysis. Furthermore, we notice two points with high leverage. Since their Cook's distance is rather small, we tolerate them.

We check for high multicollinearity of the predictors by plotting the pairwise correlations and by computing the VIFs. **Python** code:

[1]:
```python
# checking multicollinearity
import seaborn as sns

fig = sns.pairplot(x)
plt.show()
```



Python code:

```
[1]: # VIF Analysis
import numpy as np
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor

VIF = []

for i in range(1,X.shape[1]):
    VIF.append(variance_inflation_factor(X, i))


pd.DataFrame(data = {'Predictor': x.columns, 'VIF': np.round(VIF, 7)})

[j for j, val in enumerate(VIF) if val >= 10]

x.columns[[4,7]]
```

```
Index(['ED', 'LSE'], dtype='object')
```

The VIF of the variables **ED** and **LSE** are larger than 10 and thus beyond the threshold of what can be tolerated.

c) We noticed that there is a problem in the data due to multicollinearity. The possible therapeutic measures are limited in this case: since the **FoHF** can invest in all of the given subindices, we cannot amputate some of them. Similarly, creating new variables in this context does not make sense and we cannot transform the predictors either - the **FoHF** invests in the subindices which contribute directly and linearly to the return of the **FoHF**. The multicollinearity problem is caused by some highly correlated subindices.

It is possible, however, to perform a variable selection which will hopefully alleviate the multicollinearity problem. From the summary output, we conclude that the **FoHF** does not invest in all subindices. In the following subproblem, we shall try to achieve a reduction of the model size so that the final model will only contain those subindices the **FoHF** does in fact invest in.

d) i) Stepwise variable selection, starting with the full model (backward stepwise model selection). **Python** code:

```
[1]: # Define the empty dataframe and the empty predictor
results_backward = pd.DataFrame(data = {'Worst_Pred' : [], 'BIC' : []})

# Define the full predictor
x_temp = x.copy() # Copy the full set of predictors x (_temp <-> temporarily)
```

```
for j in range(x.shape[1]):
    results_j, worst_j = LMS_def.drop_one(x_temp, y, scoreby = 'BIC')

    # Update the empty predictor with the best predictor
    #x_temp = x_temp.drop(columns = worst_j)

    # Remove the chosen predictor from the list of options
    x_temp = x_temp.drop(columns = worst_j)

    # Save results
    results_backward.loc[j, 'Worst_Pred'] = worst_j
    results_backward.loc[j, 'BIC'] = results_j['BIC'].min()


print('Worst Predictors and corresponding BIC:\n', results_backward,
      '\n\nThe best model thus contains',
      x.shape[1] - results_backward['BIC'].argmin(),
      ' predictors')
```

```
Worst Predictors and corresponding BIC:
    Worst_Pred         BIC
0          DS -651.806109
1          MA -656.341582
2         EMN -659.862235
3          EM -662.473616
4          SS -665.651174
5         LSE -667.906883
6          RV -668.341764
7          CA -668.319292
8          GM -660.280513
9         FIA -628.181412
10        CTA -586.709650
11         ED -543.797909

The best model thus contains 6  predictors
```

ii) Stepwise variable selection, starting with the empty model (forward stepwise model selection). **Python** code:

[1]:
```
# Define the empty dataframe and the empty predictor
results_forward = pd.DataFrame(data = {'Best_Pred' : [], 'BIC' : []})

# Define the empty predictor
x_forward = [np.zeros(len(y))]
x_temp = x.copy() # Copy the full set of predictors x (_temp <-> temporarily)


for j in range(x.shape[1]):
    results_j, best_j = LMS_def.add_one(x_temp, x_forward, y, scoreby = 'BIC'

    # Update the empty predictor with the best predictor
    x_forward = np.concatenate((x_forward, [x[best_j]]))
    # Remove the chosen predictor from the list of options
    x_temp = x_temp.drop(columns = best_j)

    # Save results
    results_forward.loc[j, 'Best_Pred'] = best_j
    results_forward.loc[j, 'BIC'] = results_j['BIC'].min()


print('Best Predictors and corresponding AIC:\n', results_forward,
      '\n\nThe best model thus contains', results_forward['BIC'].argmin() + 1
      ' predictors')
```

```
Best Predictors and corresponding AIC:
    Best_Pred         BIC
0          GM -633.854422
1          CA -649.972305
2         FIA -654.297976
3         CTA -657.044890
4          ED -668.341764
5          RV -667.906883
6         LSE -665.651174
7          SS -662.473616
8          EM -659.862235
9         EMN -656.341582
10         MA -651.806109
11         DS -647.246196

The best model thus contains 5  predictors
```

The two variable selection methods yield almost the same model. They both have subindices **CA**, **FIA**, **ED**, **GM** and **CTA**.

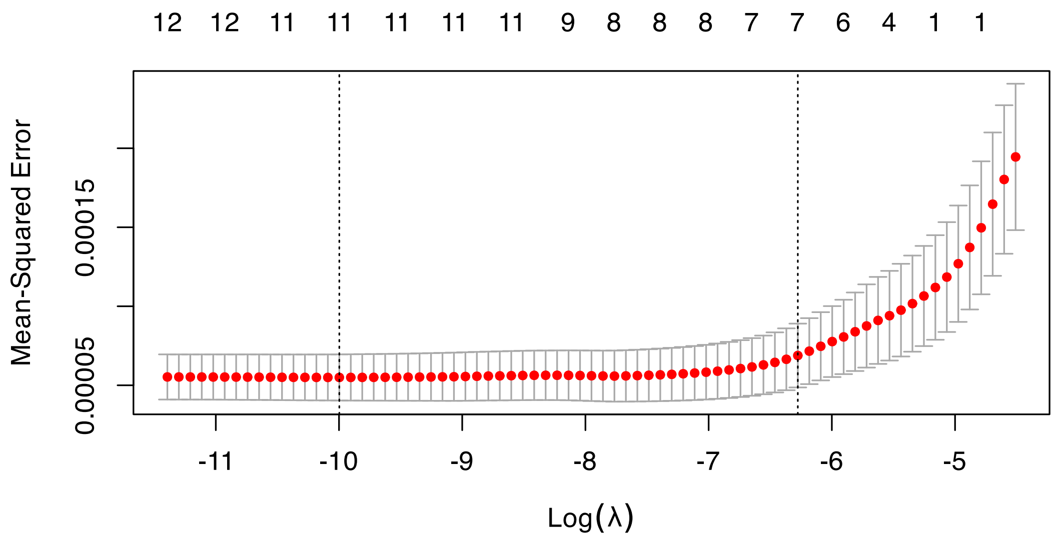
e) **Python** code:

```
[1]: # defining x_opt with optimal predictors
     x_opt = x[results_forward.iloc[:5, 0]]

     # defining design matrix (model matrix or regressor matrix)
     X_opt = sm.add_constant(x_opt)

     # Fit the linear model
     model_opt = sm.OLS(y, X_opt).fit()

     model_opt.summary()

     # ANOVA analysis
     table = sm.stats.anova_lm(model_opt, model_full)
     print(table)
```

```
   df_resid       ssr  df_diff   ss_diff         F    Pr(>F)
0      90.0  0.004003      0.0       NaN       NaN       NaN
1      83.0  0.003575      7.0  0.000428  1.419443  0.208521
```

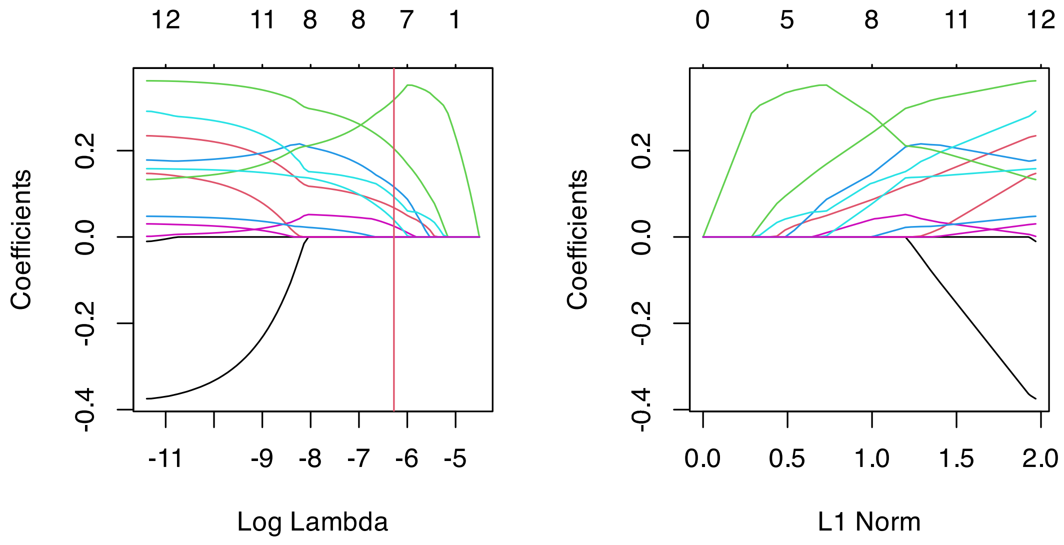
Due to the p-value of the associated F-statistic, removing those variables is indeed justified.

f)
```
## Lasso
library(glmnet)

## Lade n"otiges Paket:  Matrix

## Loaded glmnet 4.1-8

xx <- model.matrix(FoHF ~ ., data = FoHF)
yy <- FoHF$FoHF
cvfit <- cv.glmnet(xx, yy)
plot(cvfit)
```



```
coef(cvfit, s = "lambda.1se")

## 14 x 1 sparse Matrix of class "dgCMatrix"
##                          s1
## (Intercept) 0.0002196824
## (Intercept) .
## RV          .
## CA          0.0676192479
## FIA         0.2044571894
## EMN         0.1160512738
## ED          0.0927171171
## DS          0.0249532901
## MA          .
## LSE          .
## GM           0.3178869629
## EM           .
## CTA          0.0390495344
## SS           .
```

```
fit.lasso <- glmnet(xx, yy)
par(mfrow = c(1, 2))
plot(fit.lasso, label = TRUE, xvar = "lambda")
abline(v = log(cvfit$lambda.1se), col = 2)
plot(fit.lasso, label = TRUE)
```



Cross validation yields a model with 7 predictors. However, these are not identical to the ones chosen by the best BIC fit with 7 predictors. In general, the Lasso is a suitable tool as it can handle multicollinearity of predictor variables and perform variable selection. Both of these aspects are necessary here since the subindices are collinear and we know that the **FoHF** is not invested in all possible subindices. Therefore, the Lasso solution should also be considered next to the one from the variable selection with BIC.