PredMod

June 28, 2022

[2]:
```
## Config

#if you want to use this notebook and the code but would like a different setup␣
↪of the structure just change the paths here
path_data = './data/'
path_img = './img/'

#disable warnings
import warnings
warnings.filterwarnings('ignore')
```

1 Statistical Regression Analysis

Regression analysis represents a statistical method to study and model the relationship between a **response variable** and **predictor variables**. Principal goal of regression analysis is to: * predict data points based for some new values of the predictor variables (prediction) * understand how the response variable is affected by a change of the predictor variables (inference)

1.1 Error

is a random error term which is: * independent of X1, X2, . . . , Xp has mean zero

Remarks: * Error is described in terms of a probability distribution i.e. Normal Distribution * Quantity may contain unmeasured or unmeasurable variables * Simple assumption : f is linear. Linear regression allows as well to fit non-linear curves.

2 Simple Linear Regression

- 0 represents the intercept of the regression line
- 1 measures the slope of the regression line
- reads "is approximately modeled as"

2.1 Least Squares Method

Least squares approach chooses $\hat{\beta}_0$ and $\hat{\beta}_1$ to minimize the residual sum of squares (RSS)

$$RSS = \ r_1^2 + r_1^2 + ... + r_n^2$$

or equivalently

$$RSS = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + ... + (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_n)^2$$

2.1.1 Least Squares Coefficient Estimates

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{1}$$

where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{2}$$

2.2 Standard Error

Estimated deviation of the estimates to the true datapoints.

$$\operatorname{se}\left(\hat{\beta}_0\right)^2 = \sigma^2\left(\frac{1}{n} + \frac{\bar{x}^2}{\sum_{i=1}^{n} (x_i - \bar{x})^2}\right) \quad \text{and} \quad \operatorname{se}\left(\hat{\beta}_1\right)^2 = \frac{\sigma^2}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{3}$$

where $\sigma^2 = Var(\epsilon)$

Assumption that noise term is normally distributed and indipendent

2.3 Residuals and estimation of Variance

In general, σ^2 (variance of the error term) is **not** known. but can be estimated on the basis of the data. The **error term** : * cannot be observed * cannot be derived from \$ = Y - ({ }*{0} + { }*{1}X) \$ since β_0 and β_1 are unknown * Approximation for : resudials $r_i = y_i - (\hat{\beta}_0 + \hat{\beta}_1)$

Estimation of :

$$\hat{\sigma} = \text{RSE} = \sqrt{\frac{RSS}{n-2}} = \sqrt{\frac{r_1^2 + r_2^2 + ... + r_n^2}{n-2}} \tag{4}$$

- factor $1/(n-2)$ is chosen so that the estimate of turns out **unbiased**
- this estimate is known as the **residual standard error** (RSE)
- $\hat{\sigma} = RSE$

2.4 Hypothesis Test

Most common hypothesis test: * H_0: There is **no** relationship between X and Y * H_A: There is **some** relationship between X and Y Mathematically, this corresponds to testing: * H_0: \$ _{1} = 0 \$ versus * H_A: \$ _{1} 0 \$

If $\beta_1 = 0$, then model reduces to $Y = \beta_0 + \varepsilon$ and X is not associated with Y

- To test the null hypothesis, we need to determine whether $\hat{\beta}_1$ is sufficiently far from zero that we can be confident that β_1 is non-zero
  - Question: How far from 0 is far enough?
  - Answer: it depends on $se(\hat{\beta}_1)$

* if $se(\hat{\beta}_1)$ is **large**: then β_1 must be large in absolute value, to reject H_0
* if $se(\hat{\beta}_1)$ is **small**: then even relatively small values of β_1 may provide evidence that $\beta_1 \neq 0$, that is to reject H_0

2.4.1 Hypothesis Test, Test statistic and P-Value

Test-Statistic: $T = \frac{\hat{\beta}_1 - 0}{\mathrm{se}(\hat{\beta}_1)}$: measures the number of standard deviations that $\hat{\beta}_1$ is away from 0 * If there is really no relationship between X and Y , that is H_0 is **true**, then we expect that T follows a t-distribution with $n-2$ degrees of freedom * We perform an experiment and measure the realization t of the test statistic T * **p-value**: probability of observing any value of T larger than $|t|$ * If p-value is smaller than (typically $= 0.05$), then we reject H_0 and conclude: there is a relationship between X and Y * If p-value is larger than , then we retain H_0 (there is no relationship between X and Y)

3 T-Test in Linear Regression

1. **Model**:
$$Y = \beta_0 + \beta_1 X + \varepsilon, \quad \varepsilon \sim \mathcal{N}\left(0, \sigma^2\right) \tag{5}$$
2. **Null-Hypothesis**: $H_0 : \beta_1 = 0$

   Alternative Hypothesis: $H_A : \beta_1 \neq 0$ (two-sided test)
3. **Test Statistic**:
$$T = \frac{\text{observed } - \text{ expected}}{\text{estimated standard error}} = \frac{\hat{\beta}_1 - 0}{\mathrm{se}\left(\hat{\beta}_1\right)} \tag{6}$$
   Null Distribution assuming H_0 : is true: $T \sim t_{n-2}$ (the larger the degrees of freedom the more it looks like a normal distribution
4. **Significance level**:
5. **Rejection Region for Test Statistic**:
$$K = \left(-\infty, t_{n-2;\frac{\alpha}{2}}\right] \cup \left[t_{n-2;1-\frac{\alpha}{2}}, \infty\right) \tag{7}$$
   if it falls into this region, we reject H_0
6. **Test Decision**: Verify whether observed t falls into the rejection region

3.0.1 Code Example Simple Linear Regression

[4]:
```python
'''
from google.colab import drive
drive.mount('/content/drive')
'''

import pandas as pd
import statsmodels.api as sm

# Load data
```

```python
df = pd.read_csv(path_data + 'Advertising.csv')
x = df['TV']
y = df['sales']

# Linear Regression using statsmodels.api
x_sm = sm.add_constant(x)
model = sm.OLS(y, x_sm).fit()

# Now we can print a summary,
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  sales   R-squared:                       0.612
Model:                            OLS   Adj. R-squared:                  0.610
Method:                 Least Squares   F-statistic:                     312.1
Date:                Tue, 28 Jun 2022   Prob (F-statistic):           1.47e-42
Time:                        08:18:57   Log-Likelihood:                -519.05
No. Observations:                 200   AIC:                             1042.
Df Residuals:                     198   BIC:                             1049.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          7.0326      0.458     15.360      0.000       6.130       7.935
TV             0.0475      0.003     17.668      0.000       0.042       0.053
==============================================================================
Omnibus:                        0.531   Durbin-Watson:                   1.935
Prob(Omnibus):                  0.767   Jarque-Bera (JB):                0.669
Skew:                          -0.089   Prob(JB):                        0.716
Kurtosis:                       2.779   Cond. No.                         338.
==============================================================================

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

3.0.2 p-value

Hence we find for the realization of the statistic T:

$$t = \frac{\hat{\beta}_1 - 0}{\text{se}\left(\hat{\beta}_1\right)} = \frac{0.047537 - 0}{0.002691} = 17.66518 \tag{8}$$

Code Example p-value

```python
[5]: from scipy.stats import t
```

```
p_two_sided = 2 * (1 - t.cdf(17.66518, 198))
print(p_two_sided)
```

```
0.0
```

3.1 Confidence Intervals

A 95 % confidence interval is defined as a range of values such that with a 95 % probability, the range will contain the true unknown parameter. The range is defined in terms of lower and upper limits computed from the sample of data.

- for linear regression the 95% confidence interval for β_1 takes approximately the form

$$[\hat{\beta}_1 - 2 \cdot se(\hat{\beta}_1), \hat{\beta}_1 + 2 \cdot se(\hat{\beta}_1)] \tag{9}$$

---

1. The exact formula for the 95% confidence interval for the regression coefficient β_i is

$$\left[\hat{\beta}_i - t_{0.975;n-2} \cdot se(\hat{\beta}_i), \hat{\beta}_i + t_{0.975;n-2} \cdot se(\hat{\beta}_i)\right]$$

where $t_{0.975;n-1}$ is the 97.5% quantile of a t-distribution with $n-2$ degrees of freedom.

2. With **Python** we determine the 97.5% quantile of a t-distribution as follows

[6]:
```
from scipy.stats import t
import numpy as np

# the 97.5% quantile of a t-distribution:
q_975 = t.ppf(0.975, 18)
print(np.round(q_975, 4))
```

```
2.1009
```

3.1.1 Confidence Intervals Notes

Z Table vs. T Table

T Table

3.1.2 Epected value

Expected value for $\hat{y}$ for a given value of x_0 of the predictor then is $E[\hat{y}|x_0] = E[\hat{y}_0] = E[\hat{\beta}_0 + \hat{\beta}_1 \cdot x_0] = \beta_0 + \beta_1 x_0$; this corresponds to the **true** value $y_0 = f(x_0)$ which is unknown * Answer for the true value y_0, we can only determine a **confidence interval**, in this case approximate 95% confidence interval

$$[\hat{y}_0 - 2 \cdot se(\hat{y}_0), \hat{y}_0 + 2 \cdot se(\hat{y}_0)] \tag{10}$$

where

$$se(\hat{y}_0)^2 = \sigma^2 \left(\frac{1}{n} + \frac{(x_0 - \overline{x})^2}{\sum_{i=1}^{n} (x_i - \overline{x})^2} \right) \tag{11}$$

[7]:
```python
import numpy as np
import pandas as pd
import statsmodels.api as sm

# Load data
df = pd.read_csv(path_data+'Advertising.csv')
x = df['TV']
y = df['sales']

# Fit Linear Model
x = sm.add_constant(x)
model = sm.OLS(y,x).fit()

# Prediction at points 3, 100 and 270
x0 = [3, 100, 275]
x0 = sm.add_constant(x0)

predictionsx0 = model.get_prediction(x0)
predictionsx0 = predictionsx0.summary_frame(alpha=0.05)
predictionsx0 = np.round(predictionsx0, 4)
print(predictionsx0)
```

```
      mean  mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  obs_ci_upper
0   7.1752   0.4509         6.2860         8.0644        0.6879       13.6626
1  11.7863   0.2629        11.2678        12.3047        5.3393       18.2333
2  20.1052   0.4143        19.2882        20.9221       13.6273       26.5830
```

In the **Python**-output, the values

$$\hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x_0$$

can be found under **mean**; they correspond to the y-values on the regression line, thus to the predicted response given a predictor value. Under **mean ci lower** the lower limits, under **mean ci upper** the upper limits of the corresponding confidence intervals can be found.

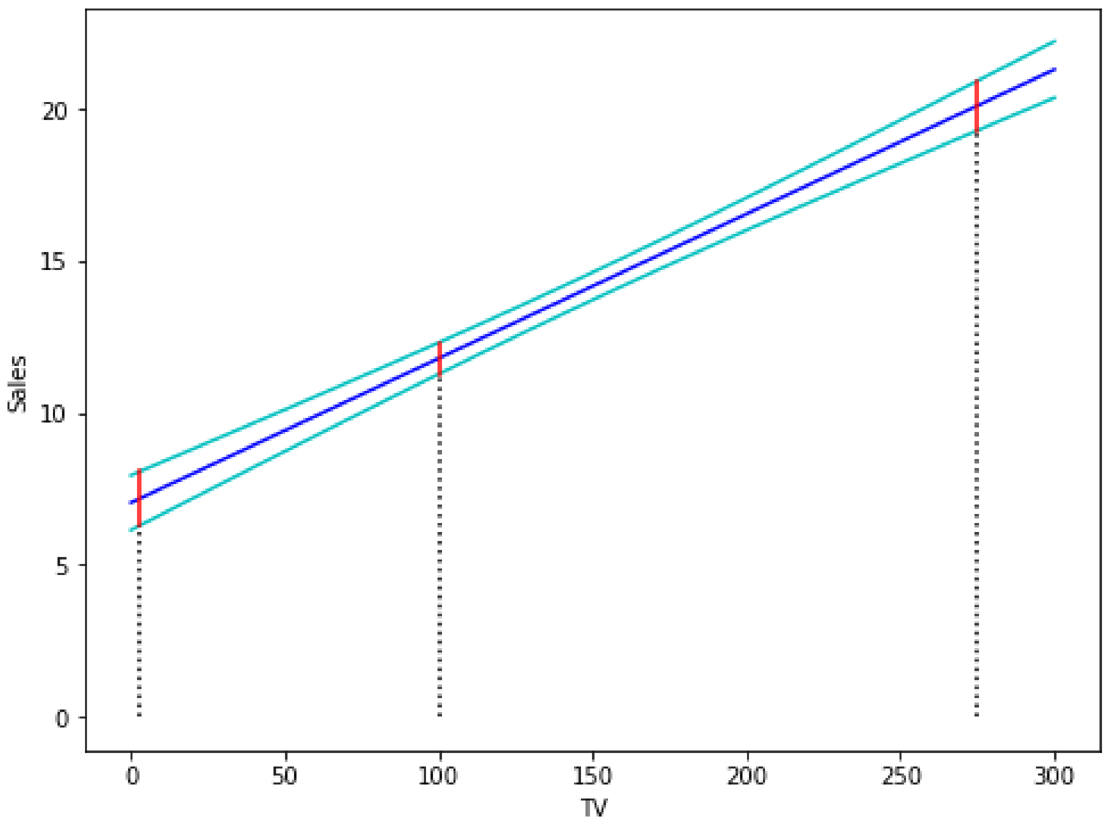
For the value $ x_{0}=100 $ the 95% confidence interval is given by

$$[11.268, 12.305]$$

The expected value of $\hat{y}_0$ given the predictor value $x_0 = 100$ is contained in this interval with a probability of 95%.

We can visualize confidence intervals for the expected response variable given an intervall of predictor values. In the next figure the regression line is plotted in blue. The green curves correspond to the lower and upper limits of the confidence intervals given a $ x_{0} $. The red lines represent the 95% confidence intervals for the values $ x_{0}=3, 100$ and $ 275 $. These intervals contain with a probability of 95% the corresponding expected values of $\hat{y}_0$.

[8]:
```
import matplotlib.pyplot as plt

x = np.linspace(0, 300)
x_sm = sm.add_constant(x)

# Predictions
predictionsx = model.get_prediction(x_sm)
# at 95% confidence:
predictionsx = predictionsx.summary_frame(alpha=0.05)

# Create figure and plot
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
# prediction
plt.plot(x, predictionsx.loc[:,'mean'], 'b-')
# upper and lower boundaries at 95%
plt.plot(x, predictionsx.loc[:,'mean_ci_lower'], 'c-')
plt.plot(x, predictionsx.loc[:,'mean_ci_upper'], 'c-')
# lines of the three points in x0:
for i in range(len(x0)):
    plt.plot([x0[i, 1], x0[i, 1]],
             [0, predictionsx0.loc[:,'mean_ci_lower'][i]], 'k:' )
    plt.plot([x0[i, 1], x0[i, 1]],
             [predictionsx0.loc[:,'mean_ci_lower'][i],
              predictionsx0.loc[:,'mean_ci_upper'][i]], 'r-' )

# Set labels and Legend
ax.set_xlabel('TV')
ax.set_ylabel('Sales')

plt.show()
```



3.2 Prediction Intervals

An important task in statistics consists of predicting future observations Y for a given value of X. Usually the variability of the fitted value Y_b is not interesting, but the variability of the future observation is. This variability is called prediction interval. There might be the temptation to use the confidence interval to indicate the interval of a future observation for a given x_0. However, this interval contains the expected value of $\hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x0$ only with a given probability. The scatter around the regression line expressed by the error term has to be accounted for as well. Thus the prediction interval can be calculated

$$se(y_0)^2 = \hat{\sigma^2}\left(1 + \frac{1}{n} + \frac{(x_0 - \overline{x})^2}{\sum_{i=1}^{n}(x_i - \overline{x})^2}\right) \tag{12}$$

- 95% **prediction interval** for future observation

$$[\hat{y}_0 - 2 \cdot se(y_0), \hat{y}_0 + 2 \cdot se(y_0)]$$

[9]:
```
print(predictionsx0)
## see example confidence interval
```

```
     mean  mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  obs_ci_upper
0  7.1752   0.4509         6.2860         8.0644        0.6879       13.6626
```

```
1  11.7863   0.2629         11.2678         12.3047          5.3393          18.2333
2  20.1052   0.4143         19.2882         20.9221         13.6273          26.5830
```

Under **mean** the predicted y-values on the regression line

$$\hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x_0$$

can be found.

obs ci lower displays the lower limits and **obs ci upper** the upper limits of the prediction intervals for the given x-values.

For the predictor value $ x_{0}=100 $ the 95% prediction interval is given by

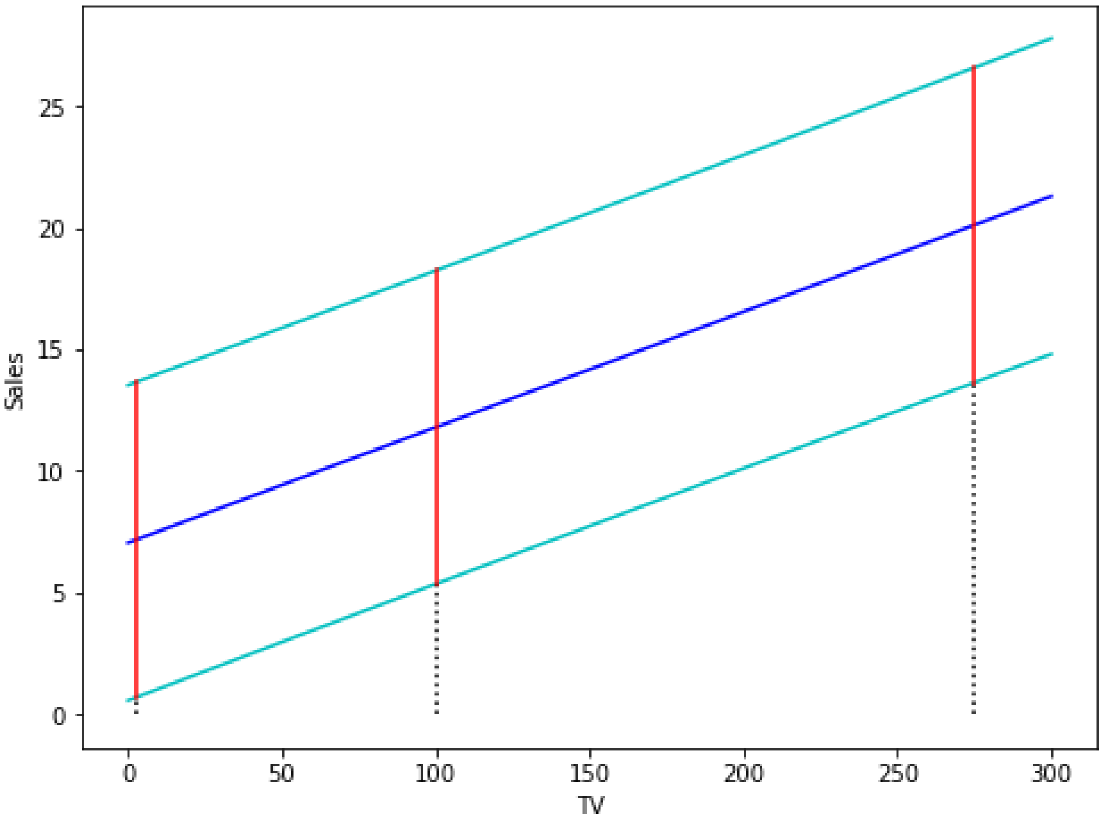
$$[5.339, 18.233]$$

A future observation $y_0 $ for given $ x_{0}=100 $ will fall with a probability of 95% into this interval. As we can observe, the prediction interval thus is clearly larger than the confidence interval for the expected value of $\hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x_0$.

It is again very instructive to visualize the point-wise prediction intervals. The following figure displays the regression line in blue. The green curves represent the upper and lower limits of the prediction intervals for future observations. The red lines correspond to 95% prediction intervals for $ x_{0}=3, 100 ,275 $. These intervals contain with a probability of 95% the true values of the corresponding future observations $ y_{0} $.

[10]:
```python
x = np.linspace(0, 300)
x_sm = sm.add_constant(x)

# Predictions
predictionsx = model.get_prediction(x_sm)
# at 95% confidence:
predictionsx = predictionsx.summary_frame(alpha=0.05)

# Create figure and plot
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
# prediction
plt.plot(x, predictionsx.loc[:,'mean'], 'b-')
# upper and lower boundaries at 95%
plt.plot(x, predictionsx.loc[:,'obs_ci_lower'], 'c-')
plt.plot(x, predictionsx.loc[:,'obs_ci_upper'], 'c-')
# lines of the three points in x0:
for i in range(len(x0)):
    plt.plot([x0[i, 1], x0[i, 1]],
             [0, predictionsx0.loc[:,'obs_ci_lower'][i]], 'k:' )
    plt.plot([x0[i, 1], x0[i, 1]],
             [predictionsx0.loc[:,'obs_ci_lower'][i],
              predictionsx0.loc[:,'obs_ci_upper'][i]], 'r-' )

# Set labels and Legend
```

```
ax.set_xlabel('TV')
ax.set_ylabel('Sales')

plt.show()
```



3.3 Model Assumptions for the Error Terms ε_i

All tests and estimation methods rely only on **model assumptions**: The eroor terms $_i$ are indipendent and nurmally distributed random variables with a constant variance: $_i$ *iid* $\mathcal{N}(0, \sigma^2)$ 1. For the expected value of all \$ _i \$ we have

$$E(_i) = 0$$

2. The error terms $_i$ all have the same constant *variace*

$$Var(_i) = \sigma^2$$

3. The error terms $_i$ are *normally distributed* 4. The error terms $_i$ are indipendent

3.4 Residual Analysis

If one or several model assumptions are violated, we should see this as a chance or starting point to adapt and/or extend our regression model to find a better and more adapted model (**explorative data analysis**)

3.4.1 RSE

The **RSE (residual standard error)** is an estimate of the standard deviation of . Roughly speaking, it is the average amount that the response will deviate from the true regression line.

$$RSE = \sqrt{\frac{r^2 + ... + r_n^2}{n-2}} = \sqrt{\frac{(y_1 - \hat{y}_1)^2 + ... + (y_n - \hat{y}_n)^2}{n-2}}$$

3.4.2 R2 Statistic

The R^2 statistic provides an alternative measure of fit

$$R^2 \equiv \frac{SQE}{SQT} = \frac{\sum \left(\hat{y}_i - \overline{y}\right)^2}{\sum \left(y_i - \overline{y}\right)^2} = 1 - \frac{SQR}{SQT} = 1 - \frac{\sum \left(y_i - \hat{y}_i\right)^2}{\sum \left(y_i - \overline{y}\right)^2} = 1 - \frac{variance\ left\ after\ regression\ fit}{total\ variance}$$

Total variance of the response variable * R^2 takes the form of a proportion - the proportion of variance explained: R^2 always takes on a value between 0 and 1, and is independent of the scale of Y * If model fits perfectly the data, then $ R^2 = 1$

3.5 Correlation coefficient

$$r_{xy} = Cor(X, Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

- is also a measure of the linear relationship between X and Y
- in simple linear regression setting:
$$r^2 = R^2$$
- Question: Why not use $r = Cor(X, Y)$ instead of R^2 in order to assess the fit of the linear model?
    - Answer: Multiple Linear Regression

3.6 Diagnostic Tool for testing Model Assumptions

The linear model assumes that there is a straight-line relationship between the predictor and the response. If f is non-linear, then model assumption $E(\varepsilon_i) = 0$ is violated.

3.6.1 Tukey-Anscombe-Plot

- we plot in the vertical axis the residuals $r_i = y_i - \hat{y}_i$
- we plot on the horizontal axis the fitted or **predicted** values $\hat{y}_i$
- we thus plot the points $(\hat{y}_i, r_i)\ for\ i = 1, ..., n$

If the residuals strongly hop around 0 means that the assumption of $E[] = 0$ is violated

Here (left) the mean value of all the residuals within the window (two gray lines) gives an idea about the nature of the data. The windows is then moved across the horizontal line

The linear model fits the data well if the points in the Tukey-Anscombe plot scatter evenly around the $r = 0$ line. To visualize the relation between the residuals r_i and the predicted response values $\hat{y}_i$, we use the smoothing approach, in particular the **LOESS** smoother.

3.6.2 Simulation of plausible Smoothing Curves

Principle idea of resampling approach : simulating data points on the basis of the existing data set. For simulated data points we fit a smoothing curve and add it to Tukey-Anscombe plot.

the conclusion, even though it does not follow the r= 0 line, since the others don't follow that line as well, it does not seem to be a systematic error

1. We keep the predicted values $\hat{y}_i$ as they are. Then, we assign to each $\hat{y}_i$ a new residual r_i^* which we obtained from sampling with replacement among the existing set of r_i
2. On the basis of the new data pairs $(\hat{y}_i, r_i^*)$, a smoothing curve is fitted, and is added to the Tukey-Anscombe plot as a grey line (the resampled data points are not shown)
3. The entire process is repeated for a number of times, e.g. one-hundred times.

3.6.3 Treatment of Case of Violation of Model Assumption: Transformation of Variables

If the Tukey-Anscombe plot indicates that there are non-linear associationn in the data (f is non-linear), then use non-linear transformations of the predictor, such as $\tilde{X} = log(X), \tilde{X} = \sqrt{X}\ or\ \tilde{X} = X^2$ in the regression model. **Solution** for previous problem : variable transformation $\tilde{X} = X^2$ establishes linear relationship between $\tilde{X}\ and\ Y$

3.6.4 Diagnostics Tool for Testing the Model Assumption $Var[\varepsilon_i]$ = constant

- Measure of scattering amplitude of errors: square root of the absolute value of the standardized residuals, that is
$$\sqrt{|\tilde{r}_i|}$$
- Standardized residuals eri are defined as follows

$$\tilde{r}_i = \frac{r_i}{\hat{\sigma}\sqrt{1 - \left(\frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_i^n (x_i - \bar{x})^2}\right)}}$$

3.6.5 Scale Location Plot

If we plot the square root of the absolute values of the standardized residuals versus the predicted values $\hat{y}_i$: Scale-Location Plot

Red curve not within grey band of simulated curves : **heteroscedasticity** this means that we cannot assume a constant variance of the error term

3.6.6 Diagnostics Tool for the Normal Distribution Assumption of the Errors $_i$

We are not able to determine the error terms ε_i directly, we use the standardized residuals instead : $\tilde{r}_i$

We check the Normal Distribution Assumption of the errors by means of a normal plot.

3.6.7 Diagnostics Tool for Independence Assumption of Errors $_i$

Example: the fact that ε_i is positive provides little or no information about the sign of $\varepsilon_i + 1$

Consequences for case of correlated error terms * The standard errors that are computed for the estimated regression coefficients or the fitted values are based on the assumption of independent error terms $_i$ * If there is correlation among the error terms, then the estimated standard errors will tend to underestimate the true standard errors. As a result, confidence and prediction intervals will be narrower than they should be

3.6.8 Outlier

An Outlier is point for which y_i is far from value $\hat{y}_i$ predicted by model.

- Red regression line : without outlier; blue regression line: with outlier
- Removing outlier: little effect on β_0 and β_0
- **BUT:** important effect on RSE and R^2 with outlier

3.6.9 Leverage Points and Leverage Statistic h_i

Leverage Statistic:

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{i'=1}^{n} (x_{i'} - \bar{x})^2}$$

Properties h_i:

- h_i increases with the distance of x_i from $\overline{x}$
- High leverage point is a point having a large distance from the center of gravity $\overline{x}$ - high momentum to turn the regression line around
- hi is always between l/n and 1
- Average leverage for all the observations is always equal to $2/n$ (simple linear regression)
- high h_i indicates a high leverage point

3.6.10 Cooks Distance

Cooks distance: measures the influence of an observation i

$$d_i = \frac{1}{\hat{\sigma}^2} \cdot \left(\underline{\hat{y}}_{(-i)} - \underline{\hat{y}}\right)^T \left(\underline{\hat{y}}_{(-i)} - \underline{\hat{y}}\right)$$

- $\underline{\hat{y}}_{(-i)}$ denotes the vector of predicted values of the ith observation is removed

3.6.11 Therapeutical Treatment in the case of $E[_i] \neq 0$

- If Tukey-Anscombe plot indicates non-linear structure in the data (f is non-linear), then a non-linear transformation of predictor such as
  - $\tilde{X} = log(X)$
  - $\tilde{X} = \sqrt{X}$
  - $\tilde{X} = X^2$

  may help to establish a linear relationship between transformed variable X and response variable Y

3.6.12 Therapeutical Treatment for $Var[_i] \neq constant$

The scattering magnitude of the residuals increases with the predicted values $\hat{y}_i$

Therapeutical Treatment: log-transformation of the response variable Y may lead to a constant variance

3.6.13 Tukey's first aid principles

- log-transformation for concentration data and absolute values
- square root transformation for count data (discrete random variables)
- arcsine-transformation $\tilde{Y} = arcsin(\sqrt{Y})$ or the

logit-transformation $\tilde{Y} = log(\frac{Y+0.005}{1.01-Y})$ for percentage data

3.6.14 Therapeutic Treatment in the Case of Outliers and High Leverage Points

Simply removing outliers or leverage points isn't a good idea an outlier should tell something about the model, since it always is potentially an indicator for a new development in the data * **Fundamental Consideration for Outliers**: an observation is considered as an outlier with respect to a given model that is not fitting this observation * **Variable transformations** may change the model so that the new model suddenly fits the observation that previously was considered an outlier

Procedure

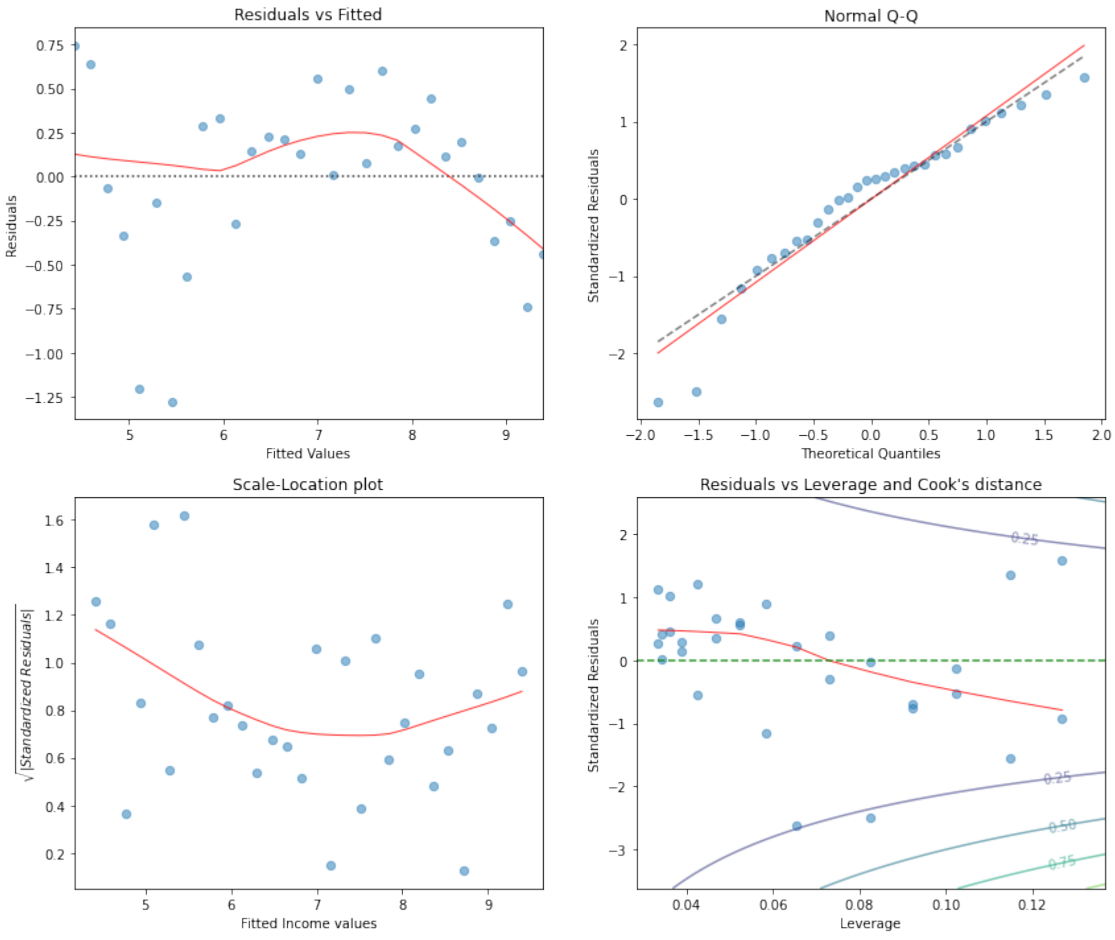
1. Check whether outlier has occured due to an error in data collection or recording
   - If an error may have occured : omit the data point
   - If an error can be excluded : go to 2
2. Attempt to transform predictor or response variables in order to make disappear the outlier. If no improvement, go to 3
3. Outlier occured due to an unusual random variation: If such outliers affect parameter estimations too much, then the observation may be removed (needs to be mentioned in the reports!)

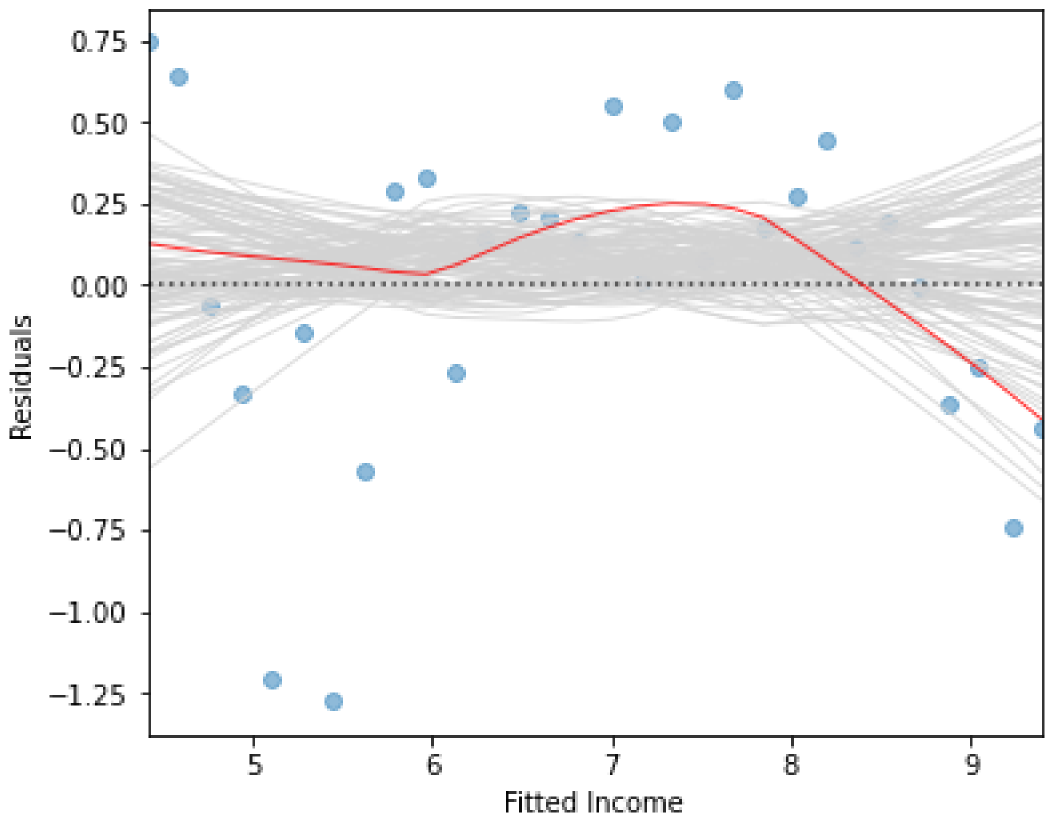
3.6.15 Code Example Testing Model Assumptions

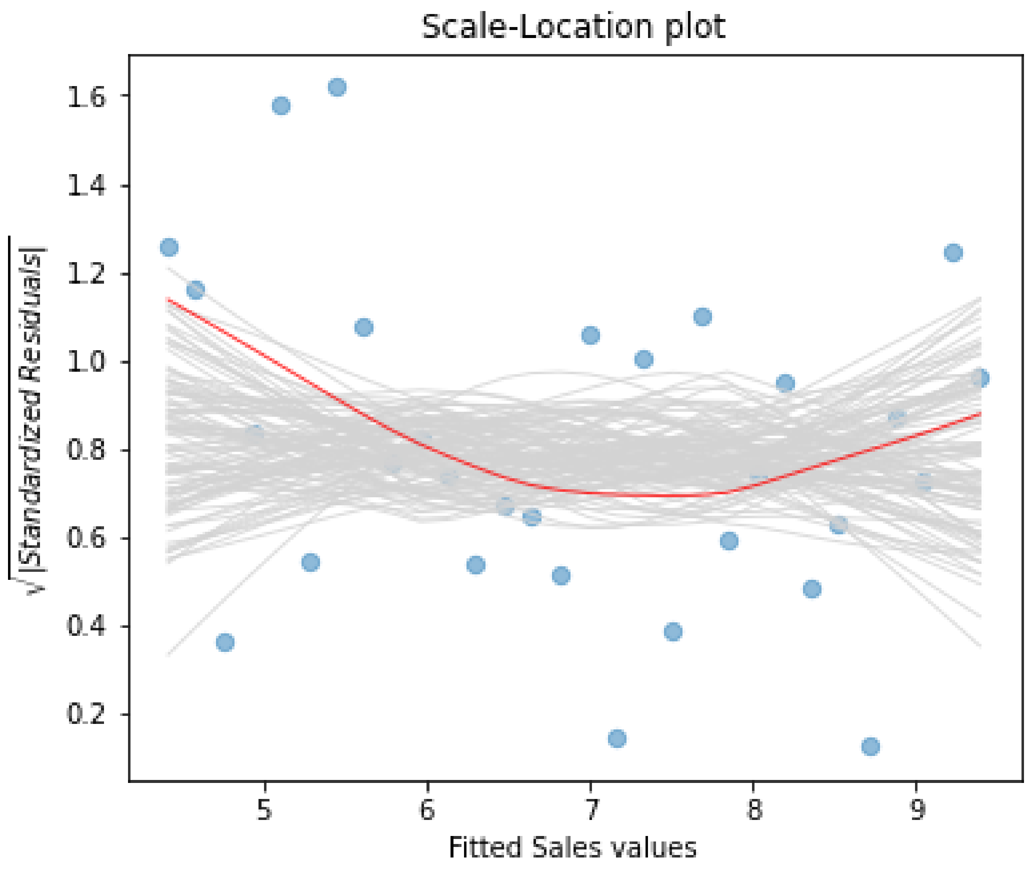
[4]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
from matplotlib import pyplot as plt
from statsmodels.graphics.gofplots import ProbPlot
from TMA_def import *

# Read data
df = pd.read_csv(path_data+'Income.csv')
x = df['education']
y = df['income']
```

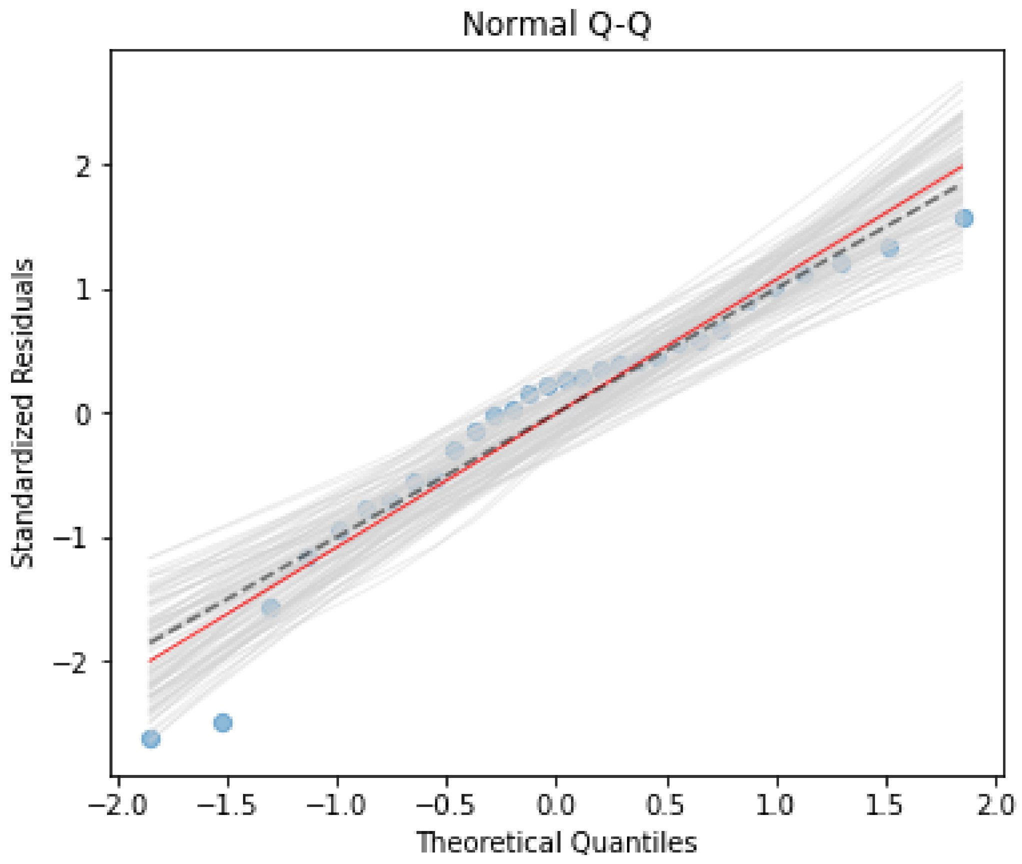
```python
""" Find Predictions, Residuals and Influence on the Residuals """
# Fit Linear Model
x_sm = sm.add_constant(x)
model = sm.OLS(np.sqrt(y), x_sm).fit()

# 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 = (14,12))

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

# Second subplot Q-Q Plot
ax2 = fig.add_subplot(2, 2, 2)
plot_QQ(ax2, res_standard)

# Third subplot: Scale location
ax3 = fig.add_subplot(2, 2, 3)
plot_scale_loc(ax3, yfit, res_stand_sqrt, x_lab='Fitted Income values')

# Fourth subplot: Cook's distance
ax4 = fig.add_subplot(2, 2, 4)
x_min, x_max = min(res_inf_leverage) - 0.005, max(res_inf_leverage) + 0.01
y_min, y_max = min(res_standard) - 1, max(res_standard) + 1
plot_cooks(ax4, res_inf_leverage, res_standard, n_pred=1,
          x_lim=[x_min, x_max], y_lim=[y_min, y_max])

# Show plot
plt.show()

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


n_samp = 100   # Number of resamples

# Create Figure and subplots
fig = plt.figure(figsize=(6,5))
ax1 = fig.add_subplot(111)

# 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

# Plot original smoothing curve
sns.residplot(x=yfit, y=res, data=df,
              lowess=True, scatter_kws={'alpha': 0.5},
              line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8})

plt.xlabel("Fitted Income")
plt.ylabel("Residuals")

# Show plots
plt.show()

#scale location plot (simulated)
n_samp = 100   # Number of resamples

# Create Figure
fig = plt.figure(figsize=(6, 5))
ax1 = fig.add_subplot(111)

# Plot Standardized Residuals using definition from TMA_def
plot_scale_loc(ax1, yfit, res_stand_sqrt, n_samp=100,
               x_lab="Fitted Sales values")



# QQ plot instance
QQ = ProbPlot(res_standard)
# Split the QQ instance in the seperate data
```

```python
qqx = pd.Series(sorted(QQ.theoretical_quantiles), name="x")
qqy = pd.Series(QQ.sorted_data, name="y")
# Estimate the mean and standard deviation
mu = np.mean(qqy)
sigma = np.std(qqy)

# Create Figure
fig = plt.figure(figsize=(6, 5))
ax1 = fig.add_subplot(111)

n_samp = 100   # Number of resamples
# 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': 0.4})

# Add plots for original data and the line x = y
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})
ax1.plot(qqx, qqx, '--k', alpha=0.5)

# Set limits and labels
plt.title('Normal Q-Q')
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Standardized Residuals')

plt.show()
```


Residuals vs Fitted
Residuals
Fitted Values
Normal Q-Q
Standardized Residuals
Theoretical Quantiles
Scale-Location plot
√|Standardized Residuals|
Fitted Income values
Residuals vs Leverage and Cook's distance
Standardized Residuals
Leverage
0.25
0.25
0.50
0.75


0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
Residuals
5
6
7
8
9
Fitted Income


Scale-Location plot
$\sqrt{|Standardized\ Residuals|}$
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
5
6
7
8
9
Fitted Sales values

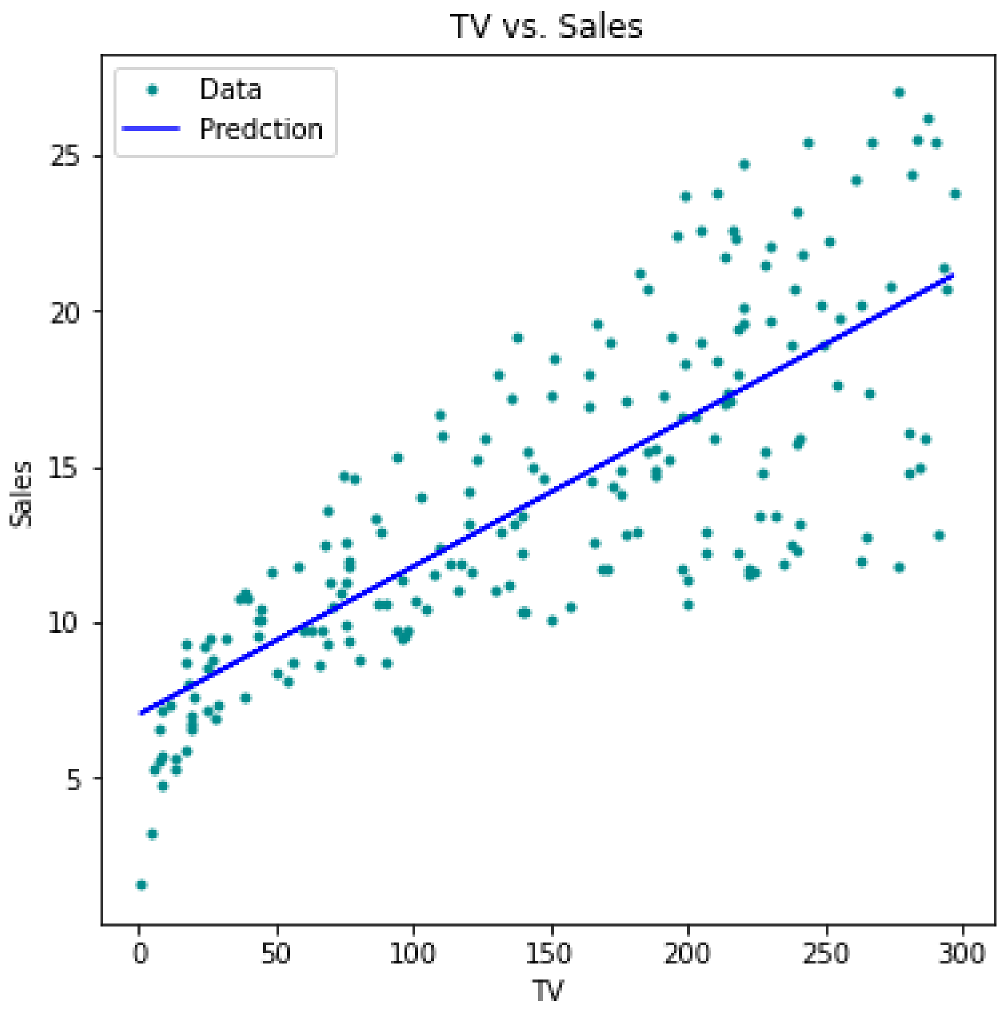


3.7 Code Example Simple Linear Regression

[7]:
```
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv(path_data+'Advertising.csv')
x = df['TV']
y = df['sales']

# Linear Regression using statsmodels.api
x_sm = sm.add_constant(x)
model = sm.OLS(y, x_sm).fit()

# Now we can print a summary,
print(model.summary())

# Predicted y
```

```python
y_pred = model.predict(x_sm)

# Create figure and plot
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(1, 1, 1)
plt.plot(df['TV'], y, marker='o', linestyle='None',
         color='darkcyan', markersize='3', label="Data")
plt.plot(df['TV'], y_pred, 'b-', label="Predction")
# Set labels and Legend
ax.set_xlabel('TV')
ax.set_ylabel('Sales')
plt.legend()
plt.title('TV vs. Sales')
plt.show()
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  sales   R-squared:                       0.612
Model:                            OLS   Adj. R-squared:                  0.610
Method:                 Least Squares   F-statistic:                     312.1
Date:                Tue, 28 Jun 2022   Prob (F-statistic):           1.47e-42
Time:                        18:32:28   Log-Likelihood:                -519.05
No. Observations:                 200   AIC:                             1042.
Df Residuals:                     198   BIC:                             1049.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          7.0326      0.458     15.360      0.000       6.130       7.935
TV             0.0475      0.003     17.668      0.000       0.042       0.053
==============================================================================
Omnibus:                        0.531   Durbin-Watson:                   1.935
Prob(Omnibus):                  0.767   Jarque-Bera (JB):                0.669
Skew:                          -0.089   Prob(JB):                        0.716
Kurtosis:                       2.779   Cond. No.                         338.
==============================================================================

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



4 Multiple Linear Regression

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p +$$

- X_j: jth predictor variable
- β_k: association between X_j and response variable Y

Parameter Estimation: Minimize RSS

$$\text{RSS} = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \left(y_i - \hat{\beta}_1 x_{i1} - \hat{\beta}_2 x_{i2} - ... - \hat{\beta}_p x_{ip} \right)^2$$

4.1 Code Beispiel

[12]:
```python
import pandas as pd
import statsmodels.api as sm

# Load data
df = pd.read_csv(path_data+'Advertising.csv')
x = df[['TV', 'radio', 'newspaper']]
y = df['sales']

# Fit Model:
x_sm = sm.add_constant(x)
model = sm.OLS(y, x_sm).fit()

# Print Model parameters
print(model.params)
```

```
const        2.938889
TV           0.045765
radio        0.188530
newspaper   -0.001037
dtype: float64
```

As we have seen, the regression coefficient estimates seem to be very different for some of the predictors with regard to the linear regression. In order to investigate this, we check how the predictors are correlated, using the DataFrame.corr() method.

[13]:
```python
# Save the Sales data in a fitting dataframe:
df = df[['TV', 'radio', 'newspaper', 'sales']]

# Print the correlation coefficients
print(df.corr())
```

```
                 TV     radio  newspaper     sales
TV         1.000000  0.054809   0.056648  0.782224
radio      0.054809  1.000000   0.354104  0.576223
newspaper  0.056648  0.354104   1.000000  0.228299
sales      0.782224  0.576223   0.228299  1.000000
```

- Is at least one of the predictors $X_l, ..., X_p$ useful in predicting the response?
- Do all the predictors $X_l, ..., X_p$ help to explain Y, or is only a subset
- How well does the model fit the data?

  R-Squared and residual standard error
- Given a set of predictor values, what response value should we predict, and how accurate is our prediction?

4.2 Hypothesis Test

Is there a Relationship Between the Response and Predictors? * **Simple Linear Regression**: If $\beta_1 = 0$, then there is no relationship between predictor and response variable, otherwise we would conclude, that there is a relationship * **Multiple Linear Regression** : Are all regression coefficients - with exception of β_0 - zero? We test the null hypothesis:

$$H_0 : \beta_1 = \beta_2 = ... = \beta_p = 0$$

versus the alternative hypothesis H_A : at least one β_i is non-zero

4.2.1 Hypothesis Test using the F-Statistic

The hypothesis test is performed by computing the **F-statistic**

$$F = \frac{(TSS - RSS)/p}{(RSS/(n-p-1))}$$

where $TSS = \sum_{i=1}^{n} (y_i - \overline{y}_i)^2$ and $RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$ and n=points and p=number of predictor variables * If linear model assumptions are **correct**, it can be shown

$$E\left(\frac{RSS}{n-p-1}\right) = \sigma^2$$

(theoretical expected value) * Provided the null hypothesis is **true** i.e. linear model assumptions are **incorrect**, then it can be shown

$$E\left(\frac{TSS - RSS}{p}\right) = \sigma^2$$

- if there is **no** relationship between response and predictors:

  the value of F-statistic is approximately **1**
- if H_A is **true**, then

  $$E\left(\frac{TSS - RSS}{p}\right) > \sigma^2$$

  and we expect F to be greater than 1
- When H_0 is true and the errors ε_i follow a normal distribution, the F-statistic follows an F-**distribution** with *p* and *n — p — 1* degrees of freedom
- For any given value of n and p, using the F-distribution the p-value associated with the F-statistic can be computed

4.3 How well does the model fit the data?

- RSE: **Residual Standard Error**
- R^2: **fraction of variance** explained by the model
  - in multiple linear regression: $R^2 = Cor(Y, \hat{Y})^2$: the square of the correlation between response and predicted response
  - a R^2 of 1 = large fraction of the variance is explained

4.4 F-Test for Subset consisting of q predictors - ANOVA

- **F-Statistic**:

$$F = \frac{(\mathrm{RSS}_0 - \mathrm{RSS})/q}{\mathrm{RSS}/(n-p-1)}$$

- **Distribution** of the test statistic F assuming H_0 is **true**

$$F \sim \mathcal{F}_{q,n-p-1}$$

4.4.1 Code Example

[14]:
```
import pandas as pd
import statsmodels.api as sm

# Load data
df = pd.read_csv(path_data+'Advertising.csv')
x1 = df[['TV', 'radio']]
x2 = df[['TV', 'radio', 'newspaper']]
y = df['sales']

# Fit model
x1_sm = sm.add_constant(x1)
x2_sm = sm.add_constant(x2)
model1 = sm.OLS(y, x1_sm).fit()
model2 = sm.OLS(y, x2_sm).fit()

# Table and print results
table = sm.stats.anova_lm(model1, model2)
print(table)
```

```
   df_resid         ssr  df_diff   ss_diff         F    Pr(>F)
0     197.0  556.913980      0.0       NaN       NaN       NaN
1     196.0  556.825263      1.0  0.088717  0.031228  0.859915
```

The difference between RSS and RSS_0 can be found in the **Python**-output under **ss_diff** and is 0.088717. The value of q is displayed under Df and is given here by 1. For the **large** model, we have

$$n - p - 1 = 200 - 3 - 1 = 196$$

degrees of freedom (**df_resid**), contrary to the **small** model that has

$$n - p - 1 = 200 - 2 - 1 = 197$$

degrees of freedom. Thus, the value of the F-statistic is (**F**)

$$
\begin{aligned}
F &= \frac{(\mathrm{RSS}_0 - \mathrm{RSS})/q}{\mathrm{RSS}/(n-p-1)} \\
&= \frac{(556.91 - 556.83)/1}{556.83/(200-3-1)} \\
&= \frac{0.088717}{556.83/196} \\
&= 0.0312
\end{aligned}
$$

The one-sided p-value in upwards direction for the $ F $-statistic assuming the null hypothesis is true, that is $\beta_3 = 0$, is displayed in the **Python**-output under **Pr(>F)** : 0.8599.

Since this p-value is significantly larger than the significance level $\alpha = 0.05$, there is no evidence to reject the null hypothesis. We conclude that the predictor **newspaper** is redundant, and we therefore can omit it.

4.4.2 Code example of ANOVA of all predictor variables

[15]:
```python
import statsmodels.formula.api as smf

# Load data
TV = df[['TV']]
radio = df[['radio']]
newspaper = df[['newspaper']]
sales = df['sales']

# Fit model using formula:
model_f = smf.ols(formula='sales ~ TV + radio + newspaper', data=df).fit()

# Table and print results
table_f = sm.stats.anova_lm(model_f, typ=2)
print(table_f)
```

```
                sum_sq     df            F        PR(>F)
TV         3058.010016    1.0  1076.405837  1.509960e-81
radio      1361.736549    1.0   479.325170  1.505339e-54
newspaper     0.088717    1.0     0.031228  8.599151e-01
Residual    556.825263  196.0          NaN           NaN
```

4.5 Factor Variables

Goal : We wish to investigate differences in credit card balance between (gender) males und females

Solution: based on the gender variable, we create a new variable * **Indicator** or dummy **variable**:

$$
x_i = \begin{cases} 1 & \text{if ith person is female} \\ 0 & \text{if ith person is male} \end{cases}
$$

- Regression Model:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i = \begin{cases} \beta_0 + \beta_1 + \varepsilon_i & \text{if ith person is female} \\ \beta_0 + \varepsilon_1 & \text{if ith person is male} \end{cases}$$

  - β_0: average credit card balance among males
  - $\beta_0 + \beta_1$: average credit card balance among females
  - β_1: average difference in credit card balance between females and males

4.5.1 Code Example for 2 factor variables

[16]:
```
import pandas as pd
import numpy as np
import statsmodels.api as sm

# Load data
df = pd.read_csv(path_data+'Credit.csv')

balance = df['Balance']

# Initiate dummy variable with zeros:
gender = np.zeros(len(balance))
# Make 1 for Female:
indices_Fem = df[df['Gender']=='Female'].index.values
gender[indices_Fem] = 1

# Fit model
gender_sm = sm.add_constant(gender)
model = sm.OLS(balance, gender_sm).fit()

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

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Balance   R-squared:                       0.000
Model:                            OLS   Adj. R-squared:                 -0.002
Method:                 Least Squares   F-statistic:                    0.1836
Date:                Tue, 28 Jun 2022   Prob (F-statistic):              0.669
Time:                        08:39:38   Log-Likelihood:                -3019.3
No. Observations:                 400   AIC:                             6043.
Df Residuals:                     398   BIC:                             6051.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        509.8031     33.128     15.389      0.000     444.675     574.931
```

```
x1            19.7331     46.051      0.429      0.669     -70.801     110.267
==============================================================================
Omnibus:                       28.438   Durbin-Watson:                   1.940
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               27.346
Skew:                           0.583   Prob(JB):                     1.15e-06
Kurtosis:                       2.471   Cond. No.                         2.66
==============================================================================

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

4.5.2 Example Multiple Linear Regression With 3 Factor Variables

For example, for the **ethnicity** variable which has *three* levels we create *two* dummy variables. The first could be

$$x_{i1} = \begin{cases} 1 & \text{if } i\text{th person is Asian} \\ 0 & \text{if } i\text{th person is not Asian} \end{cases}$$

and the second could be

$$x_{i2} = \begin{cases} 1 & \text{if } i\text{th person is Caucasian} \\ 0 & \text{if } i\text{th person is not Caucasian} \end{cases}$$

Then both of these variables can be used in the regression equation, in order to obtain the model

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \epsilon_i = \begin{cases} \beta_0 + \beta_1 + \epsilon_i & \text{if } i\text{th person is Asian} \\ \beta_0 + \beta_2 + \epsilon_i & \text{if } i\text{th person is Caucasian} \\ \beta_0 + \epsilon_i & \text{if } i\text{th person is Afro-American} \end{cases} \quad (13)$$

Now $ *{0} $ can be interpreted as the average credit card balance for African Americans, $* {1} $ can be interpreted as the difference in the average balance between the Asian and African American categories, and $ _{2} $ can be interpreted as the difference in the average balance between the Caucasian and African American categories.

- There will always be one fewer dummy variable than the number of levels.
- The level with no dummy variable African American in the example - is known as the *baseline*.
- The equation

  $$y_i = \beta_0 + \beta_1 + \beta_2 + \epsilon_i$$

  does not make sense, since this person would be Asian *and* Caucasian.

From the summary below, we see that the estimated **balance** for the baseline, African American, is 531.00.

[17]:
```
# Initiate dummy variable with zeros:
ethnicity = np.zeros((len(balance),2))
# Find indices
indices_Asi = df[df['Ethnicity'] == 'Asian'].index.values
```

```python
indices_Cau = df[df['Ethnicity'] == 'Caucasian'].index.values
# Set values
ethnicity[indices_Asi, 0] = 1
ethnicity[indices_Cau, 1] = 1

# Fit model
ethnicity_sm = sm.add_constant(ethnicity)
model = sm.OLS(balance, ethnicity_sm).fit()

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

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Balance   R-squared:                       0.000
Model:                            OLS   Adj. R-squared:                 -0.005
Method:                 Least Squares   F-statistic:                   0.04344
Date:                Tue, 28 Jun 2022   Prob (F-statistic):              0.957
Time:                        08:40:02   Log-Likelihood:                -3019.3
No. Observations:                 400   AIC:                             6045.
Df Residuals:                     397   BIC:                             6057.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        531.0000     46.319     11.464      0.000     439.939     622.061
x1           -18.6863     65.021     -0.287      0.774    -146.515     109.142
x2           -12.5025     56.681     -0.221      0.826    -123.935      98.930
==============================================================================
Omnibus:                       28.829   Durbin-Watson:                   1.946
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               27.395
Skew:                           0.581   Prob(JB):                     1.13e-06
Kurtosis:                       2.460   Cond. No.                         4.39
==============================================================================

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

4.6 Interaction Term

Having an Additivity in the model means that the effect of changes in a redictor X_j on the response Y is independent of the values of the other predictors. If the changes on one predictor have an effect on another predictor, then there is an interaction effect.

4.6.1 Code Example Interaction Term

[18]:
```python
import pandas as pd
import statsmodels.api as sm

# Load data
df = pd.read_csv(path_data + 'Advertising.csv')

# Define the linear model:
x = pd.DataFrame({
    'TV' : df['TV'],
    'radio' : df['radio'],
    'TV*radio' : df['TV'] * df['radio']})
y = df['sales']

# Fit model
x_sm = sm.add_constant(x)
model =sm.OLS(y, x_sm).fit()

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

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  sales   R-squared:                       0.968
Model:                            OLS   Adj. R-squared:                  0.967
Method:                 Least Squares   F-statistic:                     1963.
Date:                Tue, 28 Jun 2022   Prob (F-statistic):          6.68e-146
Time:                        08:41:02   Log-Likelihood:                -270.14
No. Observations:                 200   AIC:                             548.3
Df Residuals:                     196   BIC:                             561.5
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          6.7502      0.248     27.233      0.000       6.261       7.239
TV             0.0191      0.002     12.699      0.000       0.016       0.022
radio          0.0289      0.009      3.241      0.001       0.011       0.046
TV*radio       0.0011   5.24e-05     20.727      0.000       0.001       0.001
==============================================================================
Omnibus:                      128.132   Durbin-Watson:                   2.224
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1183.719
Skew:                          -2.323   Prob(JB):                    9.09e-258
Kurtosis:                      13.975   Cond. No.                     1.80e+04
==============================================================================

Notes:
```

```
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.8e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The results strongly suggest that the model that includes the interaction term is superior to the model that contains only *main effects*. The p-value for the interaction term, $ TV radio $, is extremely low, indicating that there is a strong evidence for $ H_{A}:; _{3} 0 $. In other words it is clear, that the true relationship is not additive.

The $ R^{2} $ for the model, that includes in addition to the predictors **TV** and **radio** as well the interaction term $\text{TV} \cdot \text{radio}$, is 0.968; compared to only 0.897 for the model that predicts **sales** using **TV** and **radio** without an interaction term. This means, that

$$\frac{0.968 - 0.897}{1 - 0.897} = 0.69 = 69\%$$

of the variability in **sales** that remains after fitting the additive model has been explained by the interaction term. **Python**-output suggest that an increase in **TV** advertising of CHF 1000 is associated with increased **sales** of

$$(\hat{\beta}_1 + \hat{\beta}_3 \cdot \text{radio}) \cdot 1.000 = 19 + 1.1 \cdot \text{radio}$$

units. And an increase in **radio** advertising of CHF 1000 will be associated with an increase in **sales** of

$$(\hat{\beta}_2 + \hat{\beta}_3 \cdot \text{TV}) \cdot 1.000 = 29 + 1.1 \cdot \text{TV}$$

units.

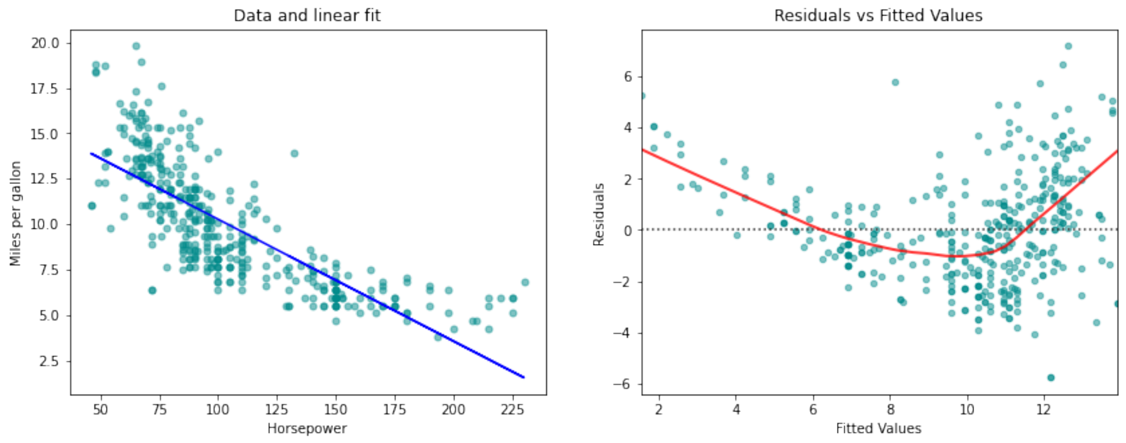
5 Non-linear Relationships and Polynomial Regression

$$Y = \beta_0 + \beta_1 \cdot X + \beta_2 \cdot X^2 + \epsilon$$

5.1 Code Example Polynomial Regression

[20]:
```python
import pandas as pd
import statsmodels.api as sm
import seaborn as sns
from matplotlib import pyplot as plt

# Load data
df = pd.read_csv(path_data+'Auto.csv')

# Define the linear model:
x = df['horsepower']
y = df['mpg']

# Fit model
x_sm = sm.add_constant(x)
model = sm.OLS(y, x_sm).fit()
```

```
# Create figure:
fig = plt.figure(figsize=(14, 5))

# Plot left figure: Scatter data and linear fit
ax1 = fig.add_subplot(1, 2, 1)
# Scatter data
plt.plot(x, y, marker='o', linestyle='None',
         color='darkcyan', markersize='5', alpha=0.5,
         label="Scatter data")
# Linear fit
plt.plot(x, model.fittedvalues, 'b-', label="Linear fit")
# Set labels
ax1.set_title('Data and linear fit')
ax1.set_xlabel('Horsepower')
ax1.set_ylabel('Miles per gallon')

# Plot right figure: Residuals vs fitted data
ax2 = fig.add_subplot(1, 2, 2)
# Residuals vs fitted value, using seaborn
ax2 = sns.residplot(
    x=model.fittedvalues, y=model.resid,
    data=df, lowess=True,
    scatter_kws={'color': 'darkcyan', 's': 20, 'alpha': 0.5},
    line_kws={'color': 'red', 'lw': 2, 'alpha': 0.8})
# Set labels
ax2.set_title('Residuals vs Fitted Values')
ax2.set_ylabel('Residuals')
ax2.set_xlabel('Fitted Values')

# Show plot
plt.show()
```



A simple approach for incorporating non-linear associations in a linear model is to include transformed versions of the predictors in the model. For example, the points in the figure seem to have a *quadratic* shape, suggesting that a model of the form

$$mpg = \beta_0 + \beta_1 \cdot horsepower + \beta_2 \cdot horsepower^2 + \epsilon$$

may provide a better fit. The equation involves predicting **mpg** using a non-linear function of **horsepower**.

But it is still a linear model!

That is, the current model is simply a multiple linear regression model with

$$X_1 = horsepower \qquad \text{and} \qquad X_2 = horsepower^2$$

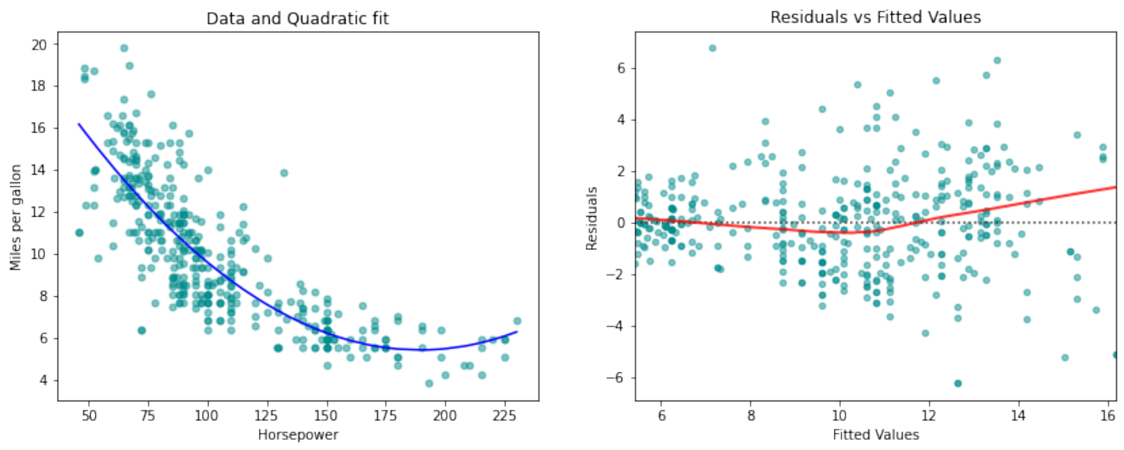
So we can use standard linear regression software to estimate β_0, β_1, and β_2 in order to produce a non-linear fit.

[21]:
```
# Define the linear model:
x = pd.DataFrame({
    'horsepower' : df['horsepower'],
    'horsepower^2' : (df['horsepower'] * df['horsepower'])})
y = df['mpg']

# Fit model
x_sm = sm.add_constant(x)
model = sm.OLS(y, x_sm).fit()

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

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.688
Model:                            OLS   Adj. R-squared:                  0.686
Method:                 Least Squares   F-statistic:                     428.0
Date:                Tue, 28 Jun 2022   Prob (F-statistic):           5.40e-99
Time:                        08:42:15   Log-Likelihood:                -797.76
No. Observations:                 392   AIC:                             1602.
Df Residuals:                     389   BIC:                             1613.
Df Model:                           2
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
const           24.1825      0.765     31.604      0.000      22.678      25.687
horsepower      -0.1981      0.013    -14.978      0.000      -0.224      -0.172
horsepower^2     0.0005   5.19e-05     10.080      0.000       0.000       0.001
==============================================================================
Omnibus:                       16.158   Durbin-Watson:                   1.078
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               30.662
Skew:                           0.218   Prob(JB):                     2.20e-07
Kurtosis:                       4.299   Cond. No.                     1.29e+05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.29e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
```

[22]:
```
# Create figure:
fig = plt.figure(figsize=(14, 5))

# Plot left figure: Scatter data and linear fit
ax1 = fig.add_subplot(1, 2, 1)
# Scatter data
plt.plot(x['horsepower'], y, marker='o', linestyle='None',
         color='darkcyan', markersize='5', alpha=0.5,
         label="Scatter data")
# Quadratic fit
x_quad_fit = x.sort_values(by='horsepower')
y_quad_fit = (model.params['const']
              + x_quad_fit['horsepower'] * model.params['horsepower']
              + x_quad_fit['horsepower^2'] * model.params['horsepower^2'])
plt.plot(x_quad_fit['horsepower'], y_quad_fit,
         'b-', label="Quadratic fit")
# Set labels
ax1.set_title('Data and Quadratic fit')
ax1.set_xlabel('Horsepower')
ax1.set_ylabel('Miles per gallon')

# Plot right figure: Residuals vs fitted data
ax2 = fig.add_subplot(1, 2, 2)
# Residuals vs fitted value, using seaborn
ax2 = sns.residplot(
    x=model.fittedvalues, y=model.resid,
    data=df, lowess=True,
    scatter_kws={'color': 'darkcyan', 's': 20, 'alpha': 0.5},
    line_kws={'color': 'red', 'lw': 2, 'alpha': 0.8})
# Set labels
ax2.set_title('Residuals vs Fitted Values')
ax2.set_ylabel('Residuals')
ax2.set_xlabel('Fitted Values')

# Show plot
```

```
plt.show()
```



The blue curve shows the resulting quadratic fit to the data. The quadratic fit appears to be substantially better than the fit obtained before, where just the linear term is included. The $ R^{2} $ of the quadratic fit is $ 0.688 $, compared to $ 0.606 $ for the linear fit, and the p-value for the quadratic term is highly significant.

Let us as well perform an ANOVA-analysis

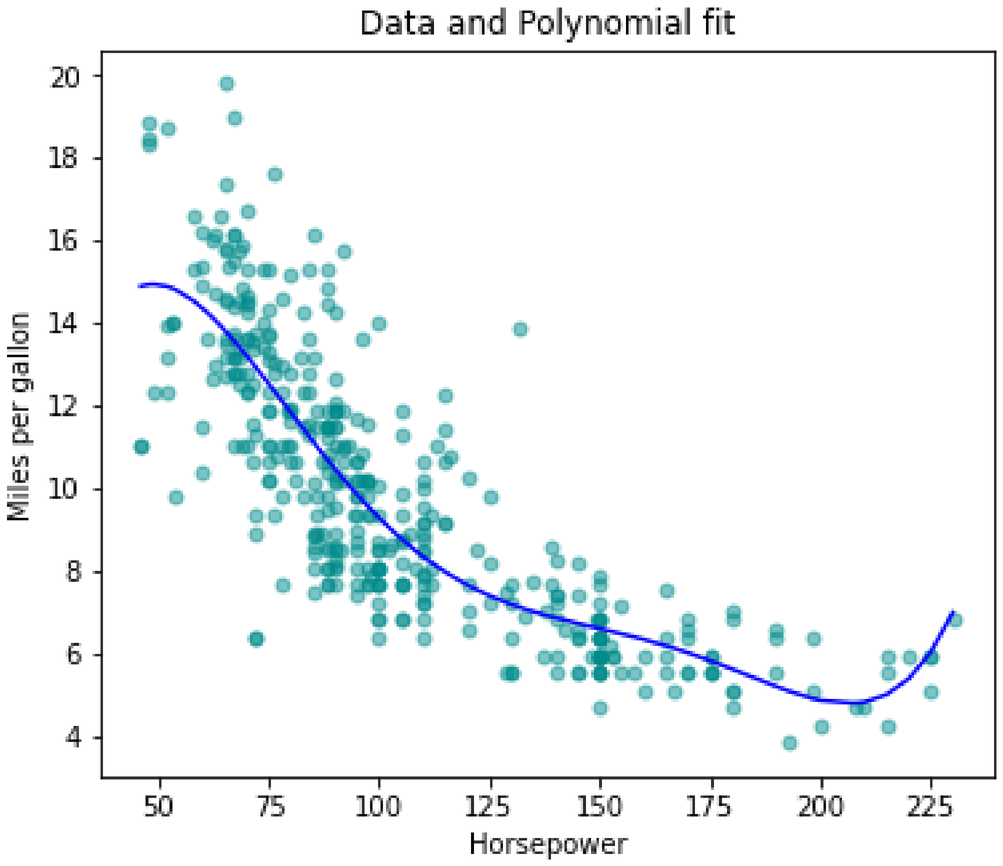
5.1.1 Anova Analysis

[23]:
```
# Define the linear model:
x = pd.DataFrame({
    'horsepower' : df['horsepower'],
    'horsepower^2' : (df['horsepower'] * df['horsepower']),
    'horsepower^3' : (df['horsepower'] ** 3),
    'horsepower^4' : (df['horsepower'] ** 4),
    'horsepower^5' : (df['horsepower'] ** 5),
    })
y = df['mpg']

# Fit model
x_sm = sm.add_constant(x)
model =sm.OLS(y, x_sm).fit()

# Print summary:
print(model.summary())
# Create figure:
fig = plt.figure(figsize=(6, 5))

# Plot left figure: Scatter data and polynomial fit
ax1 = fig.add_subplot(1, 1, 1)
```

```
# Scatter data
plt.plot(x['horsepower'], y, marker='o', linestyle='None',
         color='darkcyan', markersize='5', alpha=0.5,
         label="Scatter data")
# Polynomial fit
x_quad_fit = x.sort_values(by='horsepower')
y_quad_fit = (model.params['const']
              + x_quad_fit['horsepower'] * model.params['horsepower']
              + x_quad_fit['horsepower^2'] * model.params['horsepower^2']
              + x_quad_fit['horsepower^3'] * model.params['horsepower^3']
              + x_quad_fit['horsepower^4'] * model.params['horsepower^4']
              + x_quad_fit['horsepower^5'] * model.params['horsepower^5'])

plt.plot(x_quad_fit['horsepower'], y_quad_fit,
         'b-', label="Polynomial fit")
# Set labels
ax1.set_title('Data and Polynomial fit')
ax1.set_xlabel('Horsepower')
ax1.set_ylabel('Miles per gallon')

# Show plot
plt.show()
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.697
Model:                            OLS   Adj. R-squared:                  0.693
Method:                 Least Squares   F-statistic:                     177.4
Date:                Tue, 28 Jun 2022   Prob (F-statistic):           1.16e-97
Time:                        08:42:34   Log-Likelihood:                -791.91
No. Observations:                 392   AIC:                             1596.
Df Residuals:                     386   BIC:                             1620.
Df Model:                           5
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
const          -13.6980     12.144     -1.128      0.260     -37.575      10.179
horsepower       1.5725      0.554      2.840      0.005       0.484       2.661
horsepower^2    -0.0304      0.010     -3.170      0.002      -0.049      -0.012
horsepower^3     0.0003   7.86e-05      3.206      0.001    9.75e-05       0.000
horsepower^4 -9.695e-07   3.08e-07     -3.150      0.002   -1.57e-06   -3.64e-07
horsepower^5  1.415e-09   4.61e-10      3.068      0.002    5.08e-10    2.32e-09
==============================================================================
Omnibus:                       20.902   Durbin-Watson:                   1.113
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               40.260
Skew:                           0.305   Prob(JB):                     1.81e-09
```

```
Kurtosis:                       4.447   Cond. No.                     1.32e+13
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.32e+13. This might indicate that there are
strong multicollinearity or other numerical problems.
```



The p-value for the null hypothesis, $\beta_2 = 0$, is approximately zero. We thus reject the null hypothesis and conclude that including the quadratic term in the regression model is essential for fitting an appropriate model to the data.

If including $horsepower^2$ led to such a big improvement in the model, why not include $horsepower^3$, $horsepower^4$ or even $horsepower^5$? The Figure displays the fit that results from including all polynomials up to fifth degree in the model. The resulting fit seems unnecessarily wiggly - that is, it is unclear that including the additional terms really has led to a better fit to the data.

The approach that we have just described for extending the linear model to accomodate non-linear relationships is known as *polynomial regression*, since we have included polynomial functions of the predictors in the regression model.

5.2 Colinearity

Collinearity refers to the situation in which two or more predictor variables are closely related to one another. Collinearity can be detected by using the correlation matrix df.corr() and the *****variance inflation factor**

$$\mathrm{VIF}\left(\hat{\beta}_j\right) = \frac{1}{1 - R^2_{(X_j|X_{-j})}}$$

- close to 1, the R can not be explained by other predictor variables
- $R^2_{X_j|X_{j-1}}$ represents the R^2-value for a regression model of X_j (response variable) ontu all of the other predictors
  - $5 \leq$ **VIF** ≤ 10: indicates problematic amount of colinearity
  - **VIF** $= 1$: indicates complete absence of colinearity

5.2.1 Identification of Colinearity code example

[24]:
```python
import pandas as pd

# Load data
df = pd.read_csv(path_data+'Credit.csv')

# Drop all quantitative columns
df = df.drop(['Unnamed: 0','Gender','Student','Married','Ethnicity'],
             axis=1)

# Find the correlation Matrix using DataFrame.corr()
print(round(df.corr(), 4))
```

```
           Income   Limit  Rating   Cards     Age  Education  Balance
Income     1.0000  0.7921  0.7914 -0.0183  0.1753    -0.0277   0.4637
Limit      0.7921  1.0000  0.9969  0.0102  0.1009    -0.0235   0.8617
Rating     0.7914  0.9969  1.0000  0.0532  0.1032    -0.0301   0.8636
Cards     -0.0183  0.0102  0.0532  1.0000  0.0429    -0.0511   0.0865
Age        0.1753  0.1009  0.1032  0.0429  1.0000     0.0036   0.0018
Education -0.0277 -0.0235 -0.0301 -0.0511  0.0036     1.0000  -0.0081
Balance    0.4637  0.8617  0.8636  0.0865  0.0018    -0.0081   1.0000
```

From the **Python**-output we read off that the correlation coefficient between **limit** and **age** is $ 0.101 $ which corresponds to a rather weak correlation. On the other hand, we find for the correlation between **limit** and **rating** a value of $ 0.997 $ which is very large.

In the **Credit** data, a regression of **balance** on **age**, **rating**, and **limit** indicates that the predictors have VIF values of 1.01, 160.67, and 160.59. As we suspected, there is considerable collinearity in the data.

[30]:
```python
import numpy as np
import statsmodels.api as sm
```

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

# Define the linear model:
x = pd.DataFrame({
    'Age' : df['Age'],
    'Rating' : df['Rating'],
    'Limit' : df['Limit']})
y = df['Balance']

# VIF Analysis
x_c = sm.add_constant(x)
VIF  = []
for i in range(1,4):
    VIF.append(variance_inflation_factor(x_c.to_numpy(), i))

print(list(x.columns), '\n', np.round(VIF, 3))

# R Squared for 'complete' system
# Fit models
x_sm = sm.add_constant(x)
model = sm.OLS(y, x_sm).fit()

print('\nRsquared for the complete model is given by:\n',
      np.round(model.rsquared, 4))
```

```
['Age', 'Rating', 'Limit']
 [  1.011 160.668 160.593]

Rsquared for the complete model is given by:
 0.7536
```

When faced with the problem of collinearity, there are two simple solutions. The first is to drop one of the problematic variables from the regression. This can usually be done without much compromise to the regression fit, since the presence of collinearity implies that the information that this variable provides about the response is redundant in the presence of the other variables.

For instance, if we regress **balance** onto **age** and **limit**, without the **rating** predictor, then the resulting VIF values are close to the minimum possible value of 1, and the $ R^2 $ drops from 0.754 to 0.75.

[31]:
```
# Define the linear model:
x = x.drop('Rating', axis=1, errors='ignore')

# Fit models
x_sm = sm.add_constant(x)
model = sm.OLS(y, x_sm).fit()

# Print result
```

```
print('\n Rsquared without \'Rating\' is given by:\n',
      np.round(model.rsquared, 4))
```

```
 Rsquared without 'Rating' is given by:
 0.7498
```

So dropping **rating** from the set of predictors has effectively solved the collinearity problem without compromosing the fit.

5.3 Model Selection

1. Is there a relationship between X and Y
   - we test the null hypothesis
   - p-value associated with F-Statistic (F—Statistic) is approx. zero we **reject** null hypothesis
   - **Conclusion**: There is a relationship
2. How strong is the relationship?
   - **Model Accuracy**:
     - RSE (Residual standard error): average deviation of the response from the (true) regression line
     - R^2-value (Multiple R-squared): percentage of veriability in the response that is explained by the predictors
3. What contributes the most to Y?
   - This question is answered by considering the p-values associated with each predictor's t-statistic (t value)
4. How large is the effect of each predictor on Y
   - this question is answered by confidence intervals from the regression coefficients β_j

```
print(model.conf_int(alpha=0.05, cols=None))
```

5. How accurately can we predict $\hat{Y}$
   - We wish to predict an individual response $Y = f(X_,..., X_p) +$
     - **prediction interval**
   - We wish to predict the average response Y
     - **Confidence Interval**
6. Is the relationship linear?
   - Residual plots (in particular Tukey-Anscombe plot) showed in the case of the Advertising data a pattern that reveals a non-linear relationship
   - If the relationships are linear, then the residual plots should display no pattern
   - Solution: Taking interaction effects into account
7. Is there synergy among the predictors?
   - Standard linear regression model assumes an **additive** relationship between the predictors and the response
   - An additive model is easy to interpret because the effect of each predictor on the response is unrelated to the values of the other predictors
   - Including an *interaction term* (in the Advertisment example) in the model results in a substantial increase in R^2, from around 90% to almost 97%

[39]:
```python
# Confidence interval of the predictors
print(model.conf_int())
print('\n')
#get predictions
predictionsx = model.get_prediction(x_sm)
# at 95% confidence:
print("Prediction interval for the individual predictions (obs_ci_lower/
↪upper)\n")
print(predictionsx.summary_frame(alpha=0.05).head())

#print(predictionsx.summary_frame(alpha=0.05)[['obs_ci_lower','obs_ci_upper']].
↪head())
```

```
                0          1
const -259.575644 -87.246158
Age     -3.613562  -0.969410
Limit    0.163485   0.183245


Prediction interval for the individual predictions (obs_ci_lower/upper)

          mean    mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  \
0   373.832688  18.992519     336.494205     411.171171    -80.919133
1   790.697539  22.455263     746.551448     834.843631    335.336217
2   890.450819  18.786810     853.516751     927.384887    435.732025
3  1391.756335  30.757713    1331.287980    1452.224690    934.523923
4   519.736362  14.175471     491.867989     547.604734     65.664007

   obs_ci_upper
0    828.584509
1   1246.058862
2   1345.169613
3   1848.988747
4    973.808716
```

5.3.1 Variable Selection

Forward Stepwise Selection

1. Let M0 denote the null model which contains no predictors.
2. For $k = 0, ..., p - 1$
    1. Consider all p − k models that augment the predictors in M_k with one additional predictor.
    2. Choose the best, that is having the smallest RSS or highest R^2, among these $p - k$ models and call it M_{k+1}.

3. Select a single best model among $M_0, ..., M_p$ using Mallow's C_p, Akaike's Information Criterion (**AIC**), Bayesian Information Criterion (**BIC**), or the adjusted R^2.

Code Example

[40]:
```python
from LMS_def import *
import pandas as pd
import numpy as np
import statsmodels.api as sm

# Load data
df = pd.read_csv(path_data+'Credit.csv')

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

x_full = df.drop(columns='Balance')
y = df['Balance']


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

results, best1 = add_one(x_full, x_empty, y,scoreby='RSS')

print(results[['Predictor', 'AIC', 'BIC','R2', 'RSS']],
      '\n\nBest predictor is:',  best1)
```

```
              Predictor          AIC          BIC        R2           RSS
0           Unnamed: 0  6042.696603  6050.679532  0.000037  8.433681e+07
1               Income  5945.894250  5953.877179  0.214977  6.620874e+07
2                Limit  5499.982633  5507.965562  0.742522  2.171566e+07
3               Rating  5494.781548  5502.764477  0.745848  2.143512e+07
4                Cards  6039.710202  6047.693131  0.007475  8.370950e+07
5                  Age  6042.709965  6050.692894  0.000003  8.433963e+07
6            Education  6042.685316  6050.668245  0.000065  8.433443e+07
7        Gender__Female  6042.526817  6050.509746  0.000461  8.430102e+07
8          Student__Yes  6014.932656  6022.915585  0.067090  7.868154e+07
9          Married__Yes  6042.698437  6050.681366  0.000032  8.433720e+07
10     Ethnicity__Asian  6042.672799  6050.655728  0.000096  8.433179e+07
11  Ethnicity__Caucasian  6042.706987  6050.689916  0.000011  8.433900e+07

Best predictor is: Rating
```

We now choose the *best* variable in the sense that adding this variable leads to the regression model with the lowest RSS or the highest R^2. The variable that results in the model with the lowest RSS is in this case **rating**.

Thus, we have found the model $\mathcal{M}_1$

$$balance = \beta_0 + \beta_1 \cdot rating + \epsilon$$

We now add a further predictor variable to this model by first updating the reference model and removing the chosen predictor from the set of possible predictors. Subsequently, we can run the same procedure and decide which predictor variable to add next.

```
[42]: # Update the empty predictor with the best predictor
x_1 = [df[best1]]
# Remove the chosen predictor from the list of options
x_red = x_full.drop(columns=best1, errors='ignore')

results, best2 = add_one(x_red, x_1, y)

print(results[['Predictor', 'AIC', 'R2', 'RSS']],
      '\n\nBest predictor is:',  best2)
```

```
               Predictor          AIC        R2           RSS
0            Unnamed: 0  5496.518489  0.746016  2.142103e+07
1                Income  5212.557085  0.875118  1.053254e+07
2                 Limit  5496.632982  0.745943  2.142716e+07
3                 Cards  5494.187124  0.747492  2.129654e+07
4                   Age  5484.481339  0.753545  2.078601e+07
5             Education  5496.272851  0.746171  2.140788e+07
6        Gender__Female  5496.481640  0.746039  2.141906e+07
7          Student__Yes  5372.232473  0.813849  1.569996e+07
8          Married__Yes  5494.569548  0.747250  2.131691e+07
9      Ethnicity__Asian  5496.067431  0.746302  2.139689e+07
10  Ethnicity__Caucasian  5496.772749  0.745854  2.143465e+07

Best predictor is: Income
```

We select again the variable that leads when added to the reference model to the lowest RSS. In this case, we select the predictor variable **income** which gives us the model $\mathcal{M}_2$:

$$balance = \beta_0 + \beta_1 \cdot rating + \beta_2 \cdot income + \epsilon$$

This procedure will be repeated. In particular, we will add one variable among the remaining $p-2$ variables to the model $\mathcal{M}_2$. The resulting model with the lowest RSS will become model $\mathcal{M}_3$.

```
[43]: # Update the empty predictor with the best predictor
x_2 = np.concatenate((x_1, [df[best2]]))
# Remove the chosen predictor from the list of options
x_red = x_red.drop(columns=best2, errors='ignore')

results, best3 = add_one(x_red, x_2, y)
```

```
print(results[['Predictor', 'AIC', 'R2', 'RSS']],
      '\n\nBest predictor is:',  best3)
```

```
              Predictor          AIC        R2           RSS
0           Unnamed: 0  5214.551863  0.875120  1.053240e+07
1                Limit  5210.950291  0.876239  1.043800e+07
2                Cards  5214.477534  0.875143  1.053045e+07
3                  Age  5211.113461  0.876188  1.044226e+07
4            Education  5213.765645  0.875365  1.051172e+07
5       Gender__Female  5214.521087  0.875129  1.053159e+07
6         Student__Yes  4849.386992  0.949879  4.227219e+06
7         Married__Yes  5210.930247  0.876245  1.043747e+07
8     Ethnicity__Asian  5212.042074  0.875901  1.046653e+07
9 Ethnicity__Caucasian  5213.976405  0.875299  1.051726e+07

Best predictor is: Student__Yes
```

We can automatically select the n-best predictors using SequentialFeatureSelector from **sklearn.feature__selection**. Therfore, we need to define the Linear model with **sklearn.linear__model.LinearRegression**. The chosen predictors are returned in the support_ attribute. **Note**: If *None* features are selected, the algorithm automatically choses half number of features given.

In this case, the predictor **student** turns out to be the variable that we add to model $\mathcal{M}_2$ to obtain model $\mathcal{M}_3$:

$$balance = \beta_0 + \beta_1 \cdot rating + \beta_2 \cdot income + \beta_3 \cdot student + \epsilon$$

We will end up with the following models: $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_{10}$.

But how are we going to identify the *best* model among these 11 models? We may base our decision on the value of the AIC which is listed in the **Python** output. This statistic allows us to compare different models with each other. We will later discuss the AIC in greater detail.

The procedure we have followed can also be automated using **sklear.feature__selection**. However, this is a relatively new package, which is not as flexible or extensive yet.

[44]:
```
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.linear_model import LinearRegression

# define Linear Regression Model in sklearn
linearmodel = LinearRegression()
# Sequential Feature Selection using sklearn
sfs = SequentialFeatureSelector(linearmodel, n_features_to_select=3,
                                direction='forward')
sfs.fit(x_full, y)

# Print Chosen variables
print(x_full.columns[sfs.support_].values)
```

```
['Income' 'Rating' 'Student__Yes']
```

Backward Stepwise Selection

1. Let Mp denote the full model, which contains all p predictors.
2. For \$k = p,; p — 1,;1 \$:
    - Consider all k models that contain all but one of the predictors in M_k , for a total of k—1 predictors
    - Choose the best among these k models, and call it $M_{k—l}$. Here *best* is defined as having smallest RSS or highest R^2
3. Select a single best model among M_0, ..., M_p using C_p, AIC, BIC or adjusted R^2

Code Example We begin with the **full** model, that is $\mathcal{M}_{10}$, which contains all p predictors of the Credit data set

$$\begin{aligned} \text{balance } =&\beta_0+\beta_1 \cdot \text{ income } +\beta_2 \cdot \text{ limit } +\beta_3 \cdot \text{ rating } +\beta_4 \cdot \text{ cards } \\ &+\beta_5 \cdot \text{ age } +\beta_6 \cdot \text{ education } +\beta_7 \cdot \text{ gender } +\beta_8 \cdot \text{ student } \\ &+\beta_9 \cdot \text{ married } +\beta_{10} \cdot \text{ ethnicity } +\epsilon \end{aligned}$$

Then we remove one predictor variable from the model.

[46]:
```
import pandas as pd
import numpy as np
from LMS_def import *

# Load data
df = pd.read_csv(path_data+'Credit.csv')

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

x_full = df.drop(columns='Balance')
y = df['Balance']

results, worst1 = drop_one(x_full, y, scoreby='RSS')

print(results[['Predictor', 'AIC', 'R2', 'RSS']],
      '\n\nWorst predictor is:',  worst1)
```

```
               Predictor          AIC        R2           RSS
0            Unnamed: 0  4821.370391  0.955102  3.786730e+06
1                Income  5361.643903  0.826689  1.461702e+07
2                 Limit  4853.911857  0.951296  4.107672e+06
3                Rating  4826.005381  0.954578  3.830864e+06
4                 Cards  4837.510427  0.953253  3.942650e+06
5                   Age  4825.143746  0.954676  3.822621e+06
6             Education  4820.935924  0.955150  3.782619e+06
7        Gender__Female  4821.391825  0.955099  3.786933e+06
```

```
8            Student__Yes  5214.259751  0.880104  1.011204e+07
9            Married__Yes  4821.188761  0.955122  3.785011e+06
10       Ethnicity__Asian  4821.918353  0.955040  3.791921e+06
11   Ethnicity__Caucasian  4821.043224  0.955138  3.783634e+06

Worst predictor is: Education
```

Now we remove the *least useful* variable which is the one that yields the reduced regression model with the lowest RSS or the highest R^2. This predictor represents the most *redundant* variable because its removal improves the model most significantly with respect to the RSS. In this case, this is the predictor **education**.

Thus, we obtain the model $\mathcal{M}_9$ which is given by

$$\begin{aligned} balance = {} & \beta_0 + \beta_1 \cdot income + \beta_2 \cdot limit + \beta_3 \cdot rating + \beta_4 \cdot cards \\ & + \beta_5 \cdot age + \beta_6 \cdot gender + \beta_7 \cdot student + \beta_8 \cdot married \\ & + \beta_9 \cdot ethnicity + \epsilon \end{aligned}$$

We now remove another variable from this model. To do so we first need to update the reference model which means dropping the selected predictor from the full set. Subsequently, we run the same procedure again.

[47]:
```python
# Remove the chosen predictor from the list of options
x_red1 = x_full.drop(columns=worst1, errors='ignore')

results, worst2 = drop_one(x_red1, y)

print(results[['Predictor', 'AIC', 'R2', 'RSS']],
      '\n\nWorst predictor is:',  worst2)
```

```
               Predictor          AIC        R2           RSS
0            Unnamed: 0  4819.857603  0.955047  3.791345e+06
1                Income  5359.644014  0.826689  1.461703e+07
2                 Limit  4851.966157  0.951290  4.108230e+06
3                Rating  4824.775477  0.954491  3.838246e+06
4                 Cards  4835.976056  0.953198  3.947242e+06
5                   Age  4823.685440  0.954615  3.827801e+06
6        Gender__Female  4819.862964  0.955046  3.791396e+06
7          Student__Yes  5213.014039  0.879877  1.013112e+07
8          Married__Yes  4819.758863  0.955058  3.790410e+06
9      Ethnicity__Asian  4820.411139  0.954985  3.796596e+06
10 Ethnicity__Caucasian  4819.562597  0.955080  3.788550e+06

Worst predictor is: Ethnicity__Caucasian
```

with **sklearn.feature__selector**

[48]:
```python
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.linear_model import LinearRegression
```

```
# define Linear Regression Model in sklearn
linearmodel = LinearRegression()
# Sequential Feature Selection using sklearn
sfs = SequentialFeatureSelector(linearmodel, n_features_to_select=3,
                                direction='backward')
sfs.fit(x_full, y)

# Print Chosen variables
print(x_full.columns[sfs.support_].values)
```

```
['Income' 'Limit' 'Student__Yes']
```

Forward-Backward Stepwise Feature selection in one Code

with sklearn

[49]:
```
import pandas as pd
import numpy as np
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.linear_model import LinearRegression
import statsmodels.api as sm
from LMS_def import *

# Load data
df = pd.read_csv(path_data+'Credit.csv')

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

x_full = df.drop(columns='Balance')
y = df['Balance']

# define Linear Regression Model in sklearn
linearmodel = LinearRegression()

# Predefine DataFrame
results = pd.DataFrame(data={'BIC': [], 'AIC': [], 'R2': [],
                             'R2_adj': [], 'RSS': []})

# For each number of selected features:
for i in range(1, x_full.shape[1]):
    # Sequential Feature Selection using sklearn
    sfs = SequentialFeatureSelector(linearmodel, n_features_to_select=i,
                                    direction='forward')
    sfs.fit(x_full, y)
    chosen_predictors = x_full.columns[sfs.support_].values
    # Fit a linear model using the chosen predictors: method from LMS_def.py
    results_i = fit_linear_reg(x_full[chosen_predictors], y)
    results_i = results_i.rename(str(i))
    # Save results
    results = results.append(results_i)

print(results)
```

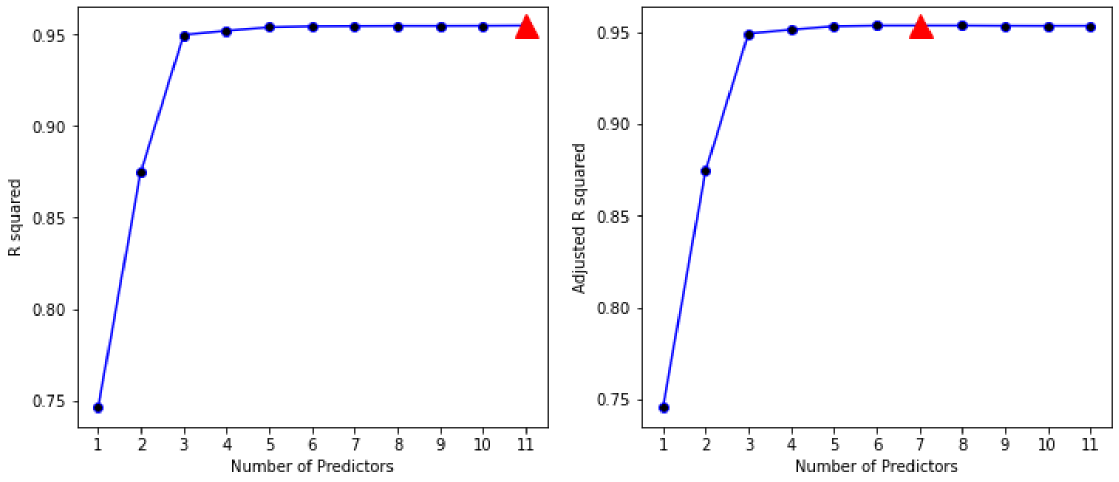
```
            BIC          AIC        R2    R2_adj           RSS
1   5502.764477  5494.781548  0.745848  0.745210  2.143512e+07
2   5224.531479  5212.557085  0.875118  0.874489  1.053254e+07
3   4865.352851  4849.386992  0.949879  0.949499  4.227219e+06
4   4852.481331  4832.524008  0.952188  0.951703  4.032502e+06
5   4841.615607  4817.666820  0.954161  0.953579  3.866091e+06
6   4842.979215  4815.038963  0.954688  0.953996  3.821620e+06
7   4847.838075  4815.906359  0.954816  0.954009  3.810814e+06
8   4852.927218  4817.004037  0.954918  0.953995  3.802227e+06
9   4858.908583  4818.993938  0.954919  0.953879  3.802131e+06
10  4864.338343  4820.432233  0.954982  0.953825  3.796796e+06
11  4869.086336  4821.188761  0.955122  0.953850  3.785011e+06
```

[50]:
```
results.sort_values(by='AIC', ascending=True)
```

[50]:
```
            BIC          AIC        R2    R2_adj           RSS
6   4842.979215  4815.038963  0.954688  0.953996  3.821620e+06
7   4847.838075  4815.906359  0.954816  0.954009  3.810814e+06
8   4852.927218  4817.004037  0.954918  0.953995  3.802227e+06
5   4841.615607  4817.666820  0.954161  0.953579  3.866091e+06
9   4858.908583  4818.993938  0.954919  0.953879  3.802131e+06
10  4864.338343  4820.432233  0.954982  0.953825  3.796796e+06
11  4869.086336  4821.188761  0.955122  0.953850  3.785011e+06
4   4852.481331  4832.524008  0.952188  0.951703  4.032502e+06
3   4865.352851  4849.386992  0.949879  0.949499  4.227219e+06
2   5224.531479  5212.557085  0.875118  0.874489  1.053254e+07
1   5502.764477  5494.781548  0.745848  0.745210  2.143512e+07
```

R^2 and adjusted R^2

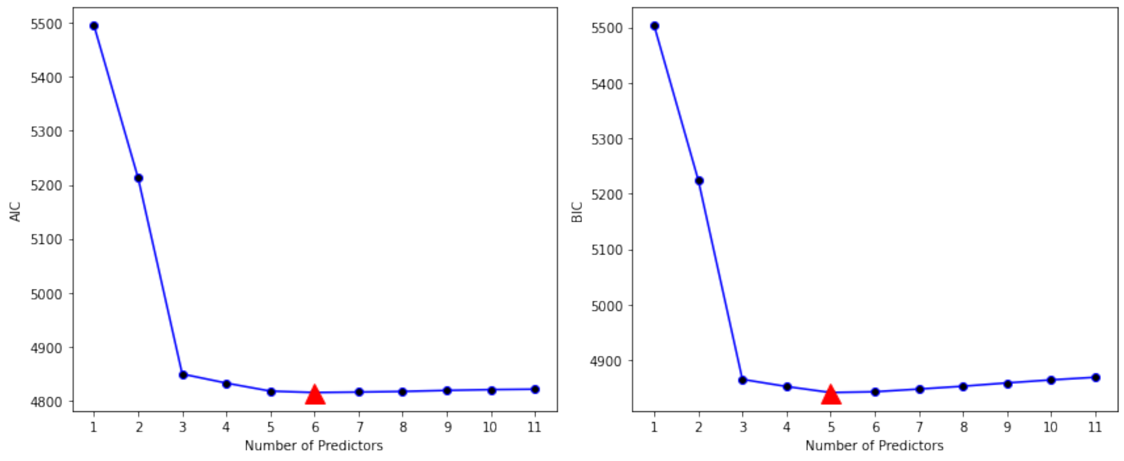
[51]:
```
import matplotlib.pyplot as plt

# Create figure
fig = plt.figure(figsize=(12, 5))

# Plot R^2, including maximum
ax1 = fig.add_subplot(1, 2, 1)
ax1.plot(results.loc[:, 'R2'], 'b-',
         marker='o', markerfacecolor='black')
ax1.plot(np.argmax(results.loc[:, 'R2']),
         np.max(results.loc[:, 'R2']),
         'r^', markersize=15)
ax1.set_xlabel('Number of Predictors')
ax1.set_ylabel('R squared')

# Plot R^2 adjusted, including maximum
ax2 = fig.add_subplot(1, 2, 2)
ax2.plot(results.loc[:, 'R2_adj'], 'b-',
         marker='o', markerfacecolor='black')
ax2.plot(np.argmax(results.loc[:, 'R2_adj']),
         np.max(results.loc[:, 'R2_adj']),
         'r^', markersize=15)
ax2.set_xlabel('Number of Predictors')
ax2.set_ylabel('Adjusted R squared')

plt.show()
```



BIC and AIC

[52]:
```
# Create figure
fig = plt.figure(figsize = (12, 5))

# Plot AIC
ax1 = fig.add_subplot(1, 2, 1)
ax1.plot(results.loc[:, 'AIC'], 'b-',
         marker='o', markerfacecolor='black')
ax1.plot(np.argmin(results.loc[:, 'AIC']),
         np.min(results.loc[:, 'AIC']),
         'r^', markersize=15)
```

```
ax1.set_ylabel('AIC')
ax1.set_xlabel('Number of Predictors')


# Plot BIC
ax2 = fig.add_subplot(1, 2, 2)
ax2.plot(results.loc[:, 'BIC'], 'b-',
         marker='o', markerfacecolor='black')
ax2.plot(np.argmin(results.loc[:, 'BIC']),
         np.min(results.loc[:, 'BIC']),
         'r^', markersize=15)
ax2.set_xlabel('Number of Predictors')
ax2.set_ylabel('BIC')

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



[53]:
```
##Add one due to Index beginning at 0
print('Number of predictors by:')
print('\t AIC: ' + str(np.argmin(results.loc[:, 'AIC']) + 1))
print('\t BIC: ' + str(np.argmin(results.loc[:, 'BIC']) + 1))
```

```
Number of predictors by:
         AIC: 6
         BIC: 5
```

using LMS_def forward

[54]:
```
import pandas as pd
import numpy as np
from LMS_def import *
```

```python
# Load data
df = pd.read_csv(path_data+'Credit.csv')

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

x_full = df.drop(columns='Balance')
y = df['Balance']

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

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

x = x0
x_red = x_full.copy()

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

    # Update the empty predictor with the best predictor
    x = np.concatenate((x, [df[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, 'BIC'] = results_i['BIC'].min()

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

```
Best Predictors and corresponding BIC:
                  Best_Pred          BIC
0                   Rating  5502.764477
1                   Income  5224.531479
2             Student__Yes  4865.352851
3                    Limit  4852.481331
4                    Cards  4841.615607
5                      Age  4842.979215
6           Gender__Female  4847.832276
7              Unnamed: 0  4852.927218
```

```
8       Ethnicity__Asian  4858.201340
9            Married__Yes  4863.468707
10  Ethnicity__Caucasian  4868.833498
11              Education  4874.337112

The best model thus contains 5  predictors
```

backward

[55]:
```python
# Load data
df = pd.read_csv(path_data+'Credit.csv')

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

x_full = df.drop(columns='Balance')
y = df['Balance']

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

# Define the full predictor
x = x_full.copy()

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

    # Update the empty predictor with the best predictor
    x = x.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',
      x_full.shape[1] - results['BIC'].argmin(), ' predictors')
```

```
Worst Predictors and corresponding BIC:
               Worst_Pred          BIC
0              Education  4868.833498
1   Ethnicity__Caucasian  4863.468707
2           Married__Yes  4858.201340
3       Ethnicity__Asian  4852.927218
4             Unnamed: 0  4847.832276
5         Gender__Female  4842.979215
6                    Age  4841.615607
```

```
7              Rating  4840.658660
8               Cards  4873.759072
9        Student__Yes  5237.176925
10             Income  5507.965562
11              Limit  6044.702777

The best model thus contains 5  predictors
```

5.3.2 Model Selection Criteria

Adjusted R^2

- **adjusted** R^2 is defined as:

$$adjusted R^2 = 1 - \frac{\frac{RSS}{(n-p-1)}}{\frac{TSS}{(n-1)}}$$

  p: # of predictor variables of least squares model

  n: # data points

- To maximise the adjusted R^2, minimise

$$\frac{RSS}{n-p-1}$$

Akaike's Information Criterion (AIC)

- AIC considers goodness-of-fit to the data and penalises the complexity of the model

$$AIC = -2\log(L) + 2q$$

  where L denotes the value of the likelihood function for a particular model and q is the number of variables of this model.

- If the errors ε in a linear regression model follow a normal distribution with expected value 0 and constant variance, then the AIC is

$$AIC = \frac{1}{n\hat{\sigma}^2}(RSS + 2p\hat{\sigma}^2)$$

  - $\hat{\sigma}$ estimated standard deviation
  - $2p\hat{\sigma}^2$ penalty term, increases if more predictors are added to the model

Mallow's C_p-statistic

- For least squares models : AIC is proportional to Mallow's C_p-statistic:

$$C_p = \frac{1}{n}(RSS + 2p\hat{\sigma}^2)$$

Bayesian information criterion BIC

- the **BIC** is defined as
$$BIC = -2\log(L) + 2\log(n)q$$
where L denotes the likelihood function for a particular model and q is the number of estimated parameters of the model

- for the least squares model with p predictors, the BIC is, up to irrelevant constants, given by
$$BIC = \frac{1}{n}(RSS + \log(n)p\hat{\sigma}^2)$$

- $\hat{\sigma}$: estimated standard deviation
- $\log(n)p\hat{\sigma}^2$ penalty term: increases BIC when more predictors are added to the model

6 Classification and Logistic Regression

- A predictive model is a functional relation between a (dependent) response variable Y and (independent) predictor variables $X_l, , X_p$
$$Y = f(X_1, ..., X_p)$$
The function f is usually unknown and has to be estimated from data

- Multiple linear regression is a methodology of prediction by means of a linear function f that is estimated via least squares optimization.

- Use of multiple linear regression, however, is limited to quantitative response variables Y, i.e., where Y is a numeric scalar.

- Classification deals with qualitative (or categorical) response variables, i.e., if Y takes on values in a finite number of classes or categories.

6.1 Simple Logistic Regression

Given a binary response variable Y and a quantitative predictor X, the simple logistic regression mode is defined as

$$P(Y = 1|X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}$$

The parameters β_0 and β_1 are called regression coefficients and are estimated from the training set

- in order to estimate the parameters, maximum likelihood method is applied

6.1.1 Code Example

[56]:
```
import numpy as np
import pandas as pd
import statsmodels.api as sm
```

```python
# Load data
df = pd.read_csv(path_data+'Default.csv', sep=';')

# Add a numerical column for default
df = df.join(pd.get_dummies(df['default'],
                            prefix='default',
                            drop_first=True))

# Fit logistic model
x = df['balance']
y = df['default_Yes']

x_sm = sm.add_constant(x)

model = sm.GLM(y, x_sm, family=sm.families.Binomial())
model = model.fit()

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

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:            default_Yes   No. Observations:                10000
Model:                            GLM   Df Residuals:                     9998
Model Family:                Binomial   Df Model:                            1
Link Function:                  Logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -798.23
Date:                Tue, 28 Jun 2022   Deviance:                       1596.5
Time:                        09:02:45   Pearson chi2:                 7.15e+03
No. Iterations:                     9   Pseudo R-squ. (CS):             0.1240
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -10.6513      0.361    -29.491      0.000     -11.359      -9.943
balance        0.0055      0.000     24.952      0.000       0.005       0.006
==============================================================================
```

From the **Python**-output we find the estimates

$\hat{\beta}_0 = -10.6513$

$\hat{\beta}_1 = 0.0055$

and thus the estimated logistic regression model for the data set **Default** is

$$\hat{p}(X) = \frac{e^{\hat{\beta}_0+\hat{\beta}_1 X}}{1+e^{\hat{\beta}_0+\hat{\beta}_1 X}} = \frac{e^{-10.6513+0.0055X}}{1+e^{-10.6513+0.0055X}} \tag{14}$$

The coefficient \$ _{1} \$ is positive and thus \$ p(X) \$ increases with \$ X \$, i.e., the larger the monthly credit card bill, the more likely the debt is not payed.

Besides the coefficients, **Python** outputs standard errors for the estimates. The \$ z \$-statistic plays an analogous role to the \$ t \$-statistic in linear regression. For instance, the $z$-statistic indicates evidence against the null hypothesis

$$\mathrm{H}_0 : \beta_1 = 0$$

In other words, we test the null hypothesis that the probability for **default=yes** does not depend on **balance**. Assuming the null hypothesis, the probability to observe the data set **Default** is given by $Pr(> |z|)$ which is negligible in our case, i.e., we reject H_0 in favor of the alternative hypothesis

$$\mathrm{H}_1 : \beta_1 \neq 0$$

and conclude that **balance** has a significant influence. Equivalently we can examine the confidence intervals for the coefficients

[57]:
```
# Find confidence interval
print(model.conf_int(alpha=0.05))
```

```
                 0         1
const   -11.359208 -9.943453
balance   0.005067  0.005931
```

The two-sided 95%-interval for β_1 is well seperated from 0 which is equivalent to rejecting H_0 with a type I error of $\alpha = 5\%$.

6.1.2 Odds

Rearranging the simple logistic regression gives

$$\frac{p(x)}{1 - p(x)} = e^{\beta_0 + \beta_1 X}$$

which is called **odds**. Odds is an equivalent way of expressing the probability of an event and is used for instance in betting agencies. Odds values close to 0 and ∞ indicate small and large probabilities, respectively. Taking the natural logarithm of the odds gives

$$\log\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 X$$

The left side of this equation is called log-odds or logit. The interpretation of the coefficients β_0 and β_1 in terms of the logit: a change of the predictor X by one unit amounts to an average change by β_1 of the logit of the response being true.

6.2 Multiple Logistic Regression

Given a binary response variable Y and predictors $X_1, ..., X_p$, the logistic regression model is defined by

$$P(Y|X_1, ..., X_p) = \frac{e^{\beta_0+\beta_1 X_1+...+\beta_p X_p}}{1 + e^{\beta_0+\beta_1 X_1+...+\beta_p X_p}}$$

The parameters $\beta_0, \beta_1, ..., \beta_p$ are called regression coefficients and are estimated from the training set.

6.2.1 Code Example

[58]:
```python
import numpy as np
import pandas as pd
import statsmodels.api as sm

# Load data
df = pd.read_csv(path_data+'Default.csv', sep=';')

# Add a numerical column for default
df = df.join(pd.get_dummies(df[['default', 'student']],
                            prefix={'default': 'default',
                                    'student': 'student'},
                            drop_first=True))
# Set ramdom seed
np.random.seed(1)
# Index of Yes:
i_yes = df.loc[df['default_Yes'] == 1, :].index

# Random set of No:
i_no = df.loc[df['default_Yes'] == 0, :].index
i_no = np.random.choice(i_no, replace=False, size=333)

# Fit Linear Model on downsampled data
i_ds = np.concatenate((i_no, i_yes))
x_ds = df.iloc[i_ds][['balance', 'income', 'student_Yes']]
y_ds = df.iloc[i_ds]['default_Yes']

# Model using statsmodels.api
x_sm = sm.add_constant(x_ds)
model_sm = sm.GLM(y_ds, x_sm, family=sm.families.Binomial())
model_sm = model_sm.fit()

print(model_sm.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:            default_Yes   No. Observations:                  666
Model:                            GLM   Df Residuals:                      662
```

```
Model Family:                Binomial   Df Model:                        3
Link Function:                  Logit   Scale:                      1.0000
Method:                          IRLS   Log-Likelihood:            -186.21
Date:                Tue, 28 Jun 2022   Deviance:                   372.42
Time:                        09:03:44   Pearson chi2:                 571.
No. Iterations:                     7   Pseudo R-squ. (CS):         0.5627
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -7.1303      0.869     -8.205      0.000      -8.833      -5.427
balance        0.0060      0.000     12.928      0.000       0.005       0.007
income     -1.454e-05    1.6e-05     -0.909      0.363   -4.59e-05    1.68e-05
student_Yes   -0.8278      0.465     -1.780      0.075      -1.739       0.084
==============================================================================
```

[59]:
```
print(model_sm.conf_int(alpha=0.05))
```

```
                    0         1
const       -8.833441 -5.427107
balance      0.005052  0.006857
income      -0.000046  0.000017
student_Yes -1.739175  0.083510
```

6.3 Model Assestment

6.3.1 Classification Error

Let $(x_1, y_1), ..., (x_n, y_n))$ be observations of (X, Y) and $\hat{y}_i = \hat{f}(x_i)$ the corresponding predictions. The classification error is given by

$$Err = \frac{1}{n}\sum_{i=1}^{n} I(y_{\neq}\hat{y}_i)$$

Here, I denotes the indicator function which takes on value 1 if its argument is true and 0 otherwise. In other words, the classification error is the proportion of cases with a wrong prediction (either false positive or false negative).

6.3.2 Confusion Matrix

6.3.3 Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- TP: True Positive
- TN: True Negative
- FN: False Negative
- TN: True Negative

in other words

$$\frac{\text{correct classifications}}{\text{all classifications}}$$

6.3.4 Precision

Ratio of correctly predicted positive observations to the total predicted positive observations

$$Precision = \frac{TP}{TP + FP}$$

6.3.5 Recall (Sensitivity)

Ratio of correctly predicted positive observations to all positive observations

$$Recall = \frac{TP}{TP + FN}$$

6.3.6 Specificity, Selectivity or True Negative Rate

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} = 1 - \mathrm{FPR}$$

6.3.7 F1 Score

Weighted average of Precision and Recall, therefore this score takes both, false positives and false negatives into account

$$\text{F1 Score} = \frac{2 \cdot (Recall \cdot Precision)}{Recall + Precision}$$

6.3.8 False Positive Rate

The false positive rate (or "false alarm rate") usually refers to the expectancy of the false positive ratio.

$$FPR = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

6.3.9 False Negative Rate

$$FNR = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}}$$

6.3.10 Code Examples

[60]:
```
import numpy as np
import pandas as pd
import statsmodels.api as sm
```

```
# Load data
df = pd.read_csv(path_data+'Default.csv', sep=';')

# Add a numerical column for default
df = df.join(pd.get_dummies(df['default'],
                            prefix='default',
                            drop_first=True))

# Fit logistic model
x = df['balance']
y = df['default_Yes']

x_sm = sm.add_constant(x)

model = sm.GLM(y, x_sm, family=sm.families.Binomial())
model = model.fit()
print(model.summary())
# Predict for balance = 1000
x_pred = [1, 1000]
y_pred = model.predict(x_pred)

print('\nPrediction with a balance of 1000: '+str(y_pred))
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:            default_Yes   No. Observations:                10000
Model:                            GLM   Df Residuals:                     9998
Model Family:                Binomial   Df Model:                            1
Link Function:                  Logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -798.23
Date:                Tue, 28 Jun 2022   Deviance:                       1596.5
Time:                        09:17:05   Pearson chi2:                 7.15e+03
No. Iterations:                     9   Pseudo R-squ. (CS):             0.1240
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -10.6513      0.361    -29.491      0.000     -11.359      -9.943
balance        0.0055      0.000     24.952      0.000       0.005       0.006
==============================================================================

Prediction with a balance of 1000: [0.00575215]
```

This probability of default is well below 1%, which is very low. However, a different individual with **balance = 2000** has a default probability of approximately 59%.

[61]:
```
# Predict for balance = 2000
x_pred = [1, 2000]
y_pred = model.predict(x_pred)

print(y_pred)
```

```
[0.58576937]
```

Classification Error

[62]:
```
##Training Error
# Predict for training data
x_pred = x_sm
y_pred = model.predict(x_pred)

# Round to 0 or 1
y_pred = y_pred.round()

# Compute training error
e_train = abs(y - y_pred)
e_train = e_train.mean()

print(e_train)
```

```
0.0275
```

The value of the training error in this example is 0.0275, which is to say that approximately 97.25% of the cases in the training set are classified correctly.

Confusion Matrix

[63]:
```
# Create confusion matrix
confusion = pd.DataFrame({'predicted': y_pred,
                          'true': y})
confusion = pd.crosstab(confusion.predicted, confusion.true,
                        margins=True, margins_name="Sum")

print(confusion)
```

```
true          0    1    Sum
predicted
0.0        9625  233   9858
1.0          42  100    142
Sum        9667  333  10000
```

F1-Scores

[64]:
```
#Find f1 score
from sklearn.metrics import f1_score
```

```python
#if we consider default_yes as positive
# Find F1-score
f1_pos = f1_score(y, y_pred, pos_label=1, average='binary')

#if we consider default_no as positive
f1_neg = f1_score(y, y_pred, pos_label=0, average='binary')

print('\nF1-Score (positive = default) = \n', f1_pos,
      '\nF1-Score (positive = not-default) = \n', f1_neg)
```

```
F1-Score (positive = default) =
 0.42105263157894746
F1-Score (positive = not-default) =
 0.9859154929577464
```

Downsampling

[65]:
```python
# Load data
df = pd.read_csv(path_data+'Default.csv', sep=';')

# Add a numerical column for default
df = df.join(pd.get_dummies(df['default'],
                            prefix='default',
                            drop_first=True))
# Set ramdom seed
np.random.seed(1)
# Index of Yes:
i_yes = df.loc[df['default_Yes'] == 1, :].index

# Random set of No:
i_no = df.loc[df['default_Yes'] == 0, :].index
i_no = np.random.choice(i_no, replace=False, size=333)

# Fit Linear Model on downsampled data
i_ds = np.concatenate((i_no, i_yes))
x_ds = df.iloc[i_ds]['balance']
y_ds = df.iloc[i_ds]['default_Yes']

x_sm = sm.add_constant(x_ds)

model_ds = sm.GLM(y_ds, x_sm, family=sm.families.Binomial())
model_ds = model_ds.fit()

print(model_ds.summary())

# Predict for downsampled data
x_pred_ds = x_sm
```

```python
y_pred_ds = model_ds.predict(x_pred_ds)

# Round to 0 or 1
y_pred_ds = y_pred_ds.round()

# Classification error on training data:
e_train = abs(y_ds- y_pred_ds)
e_train = e_train.mean()

print('\nClassification Error: ' + str(np.round(e_train, 4)))

# Create confusion matrix
confusion = pd.DataFrame({'predicted': y_pred_ds,
                          'true': y_ds})
confusion = pd.crosstab(confusion.predicted, confusion.true,
                        margins=True, margins_name="Sum")
print('\nConfusion Matrix\n')
print(confusion)

# Print F1-scores
f1_pos = f1_score(y_ds, y_pred_ds, pos_label=1, average='binary')
f1_neg = f1_score(y_ds, y_pred_ds, pos_label=0, average='binary')

print('\nF1-Score (positive = default) = \n', f1_pos,
      '\nF1-Score (positive = not-default) = \n', f1_neg)
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:            default_Yes   No. Observations:                  666
Model:                            GLM   Df Residuals:                      664
Model Family:                Binomial   Df Model:                            1
Link Function:                  Logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -188.19
Date:                Tue, 28 Jun 2022   Deviance:                       376.38
Time:                        09:18:26   Pearson chi2:                     564.
No. Iterations:                     7   Pseudo R-squ. (CS):             0.5601
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -7.7720      0.630    -12.332      0.000      -9.007      -6.537
balance        0.0059      0.000     12.850      0.000       0.005       0.007
==============================================================================

Classification Error: 0.1171

Confusion Matrix
```

```
true          0    1  Sum
predicted
0.0         293   38  331
1.0          40  295  335
Sum         333  333  666

F1-Score (positive = default) =
 0.8832335329341318
F1-Score (positive = not-default) =
 0.8825301204819278
```

6.3.11 Cross Validation

- Validating the predictive accuracy of a statistical model on the same data the model was built from is not a good idea
- Split data into test data and training data, then split training data into k folds
- 5-fold Cross-Validation

  Train data |

  Test Data |

becomes

fold 1 | fold 2 | fold 3 | fold 4 | fold 5 - Validation Data |

Test Data|

- In general: $k = 5, 10$; for $k = n$ leave-one-out cross-validation

Code Example Cross Validation

[66]:
```python
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load data
df = pd.read_csv(path_data+'Default.csv', sep=';')

# Add a numerical column for default
df = df.join(pd.get_dummies(df['default'],
                            prefix='default',
                            drop_first=True))

# Set ramdom seed
np.random.seed(1)
# Index of Yes:
i_yes = df.loc[df['default_Yes'] == 1, :].index
```

```
# Random set of No:
i_no = df.loc[df['default_Yes'] == 0, :].index
i_no = np.random.choice(i_no, replace=False, size=333)

# Fit Linear Model on downsampled data
i_ds = np.concatenate((i_no, i_yes))
x_ds = df.iloc[i_ds][['balance']]
y_ds = df.iloc[i_ds]['default_Yes']

model = LogisticRegression()

# Calculate cross validation scores:
scores = cross_val_score(model, x_ds, y_ds, cv=5)
print(scores)
print(np.mean(scores))
```

```
[0.93283582 0.84962406 0.85714286 0.90225564 0.87218045]
0.8828077656828638
```

6.4 Maximal Margin Classifier

- In p-dimensional space, a hyperplane is a flat affine subspace of dimension p—1
- In two dimensions, a hyperplane is a one-dimensional subspace — in other words: a straight line
- In three dimensions, a hyperplane is a flat two-dimensional subspace, i.e. a plane
- In two dimensions, a hyperplane is defined by the equation

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0 \quad (1)$$

  for parameters β_0, β_1 and β_2

- When we say that (1) „defines" the hyperplane, we mean that any $X = (X_1, X_2)^T$ for which (1) holds is a point on the hyperplane
- Note that (1) is simply the equation of a straight line
- Generalization to p-dimensional space:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p = 0 \quad (2)$$

- Equation (2) defines a (p—1)-dimensional hyperplane
- If a point $X = (X_l, X_2, ..., X_p)^T$ in p-dimensional space (i.e., a vector of length p) satisfies (2), then X lies on the hyperplane
- Suppose that X satisfy

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p < 0$$

  then this tells us that X lies to **one side** of the hyperplane

- if
$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p > 0$$
then X lies on the **other side** of the hyperplane.

6.4.1 Seperating Hyperplane

- has property
$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} > 0 \text{ if } y_i = 1$$
and
$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} < 0 \text{ if } y_i = -1$$
- Equivalently a separating hyperplane has the property that

$$y_i \cdot (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}) > 0 \text{ for all } i = 1, ..., n$$

* classifying **test observations** x^* based of the *sign* of

$$f(x^*) = \beta_0 + \beta_1 x_1^* + \beta_2 x_2^* + ... + \beta_p x_p^*$$

- if $f(x^*)$
  - positive: class 1
  - negative: class -1
- if $f(x^*)$ is far from 0, then there is a large confidence about the class assignment

6.4.2 Support Vector Machines

- Support vector machine (SVM) is an extension of the support vector classifier that results from enlarging the feature space
- Main idea: we may want to enlarge our feature space in order to accommodate a non-linear boundary between the classes
- Rather than fitting a support vector classifier using p features
$$X_1, X_2, ..., X_p$$
- We could instead fit a support vector classifier using $2p$ features

$$X_1, X_1^2, X_2, X_2^2, ..., X_p, X_p^2$$

- in SVM, it is always recommended to **standardize the features**

Optimization Problem

1. Maximize M
$$\max_{\beta_0, \beta_1 \cdots, \beta_p, {}_1, {}_2 \cdots, \varepsilon_p} \quad (1)$$
where are slack variables. They allow "slack" on the decision boundary, meaning some points are allowed to violate the boundary

2. subject to

$$y_i\left(\beta_0+\sum_{j=1}^{p}\beta_{j1}x_{j1}+\sum_{j=1}^{p}\beta_{j2}x_{j2}^2\right)\geq M(1-{}_i) \quad (2)$$

   for all $i=1,2...,n$
3. and subject to

$$\sum_{j=1}^{p}\sum_{k=1}^{2}\beta_{jk}^2=1 \quad (3)$$

4. and to

$${}_i\geq 0;\ \sum_{i=1}^{n}{}_i\leq C \quad (4)$$

   where C is a nonnegative tuning parameter and can be interpreted as maximum number of points lying on the opposite of the hyperplane

Kernel Trick

- a **linear support vector classifier** can be represented as

$$f\left(x^*\right)=\beta_0+\sum_{i=1}^{n}\alpha_i\left\langle x^*,x_i\right\rangle$$

  where x_i denotes *training* observations and x^* denotes *new* observations we want to classify
- Support vector machine (SVM) is an extension of the support vector classifier that results from enlarging the feature space in a specific way, using kernels
- In representing the linear classifier and in computing its coefficients, all we need are inner products
- This results in the support vector machine

$$f\left(x^*\right)=\beta_0+\sum_{i\in\mathcal{S}}\alpha_i K\left(x_i,x_{i'}\right)$$

  where

$$K\left(x_i,x_{i'}\right)$$

  is called **kernel function** and $\mathcal{S}$ is the collection of indices of these support vectors
- the advantages of using kernels is computational savings

Linear Kernel

$$K\left(x_i,x_{i'}\right)=\sum_{j=1}^{p}X_{ij}x_{i'j}$$

where p is the number of dimensions and x_i are the fitted values * **Linear kernel** essentially quantifies the similarity of a pair of observations using Pearson (standard) correlation

Polynomial kernel function

$$f\left(x^*\right)=\beta_0+\sum_{i \in \mathcal{S}} \alpha_i K\left(x^*, x_{i'}\right)$$

for $K(x_i, x_{i'})$ the **polynomial kernel function** of degree d

$$K\left(x_i, x_{i'}\right)=\left(1+\sum_{j=1}^p x_{ij} x_{i'j}\right)^d$$

where d is a positive polynomial integer

Radial Kernel

- Now, we choose in
  $$f\left(x^*\right)=\beta_0+\sum_{i \in \mathcal{S}} \alpha_i K\left(x^*, x_{i'}\right)$$
  for $K\left(x_i, x_{i'}\right)$ the radial kernel
  $$K\left(x_i, x_{i'}\right)=\exp \left(-\gamma \sum_{j=1}^p\left(x_{ij}-x_{i'j}\right)^2\right)$$
  where γ is a positive constant
- If a given test observation $x^*=\left(x_1^* \ldots, x_p^*\right)$ is far from training observation x_i wrt Euclidean distance, then
  $$\sum_{j=1}^p\left(x_j^*-x_{i'j}\right)^2$$
  will be large
- Then
  $$K\left(x_i, x_{i'}\right)=\exp \left(-\gamma \sum_{j=1}^p\left(x_{ij}-x_{i'j}\right)^2\right)$$
  will be tiny
- This means that the radial kernel has very local behavior, in the sense that only nearby training observations have an effect on the class label of a test observation

Code example SVM

- with kernel trick
- and cross validation

[67]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.model_selection import GridSearchCV
from SVM_def import SVM_def
```

```
# SVM_def is a class containing definitions used throughout this Chapter
svm_def = SVM_def()

# Create data
n = 10
np.random.seed(6)
# x from normal distribution # y is n elements -1 and n elements 1
x = np.random.normal(loc = 2, scale = 1.0, size = (2*n, 2))
y = np.concatenate((-1 * np.ones(n), np.ones(n)))

x[y==1, :] += 1      # offset x[:][:] for y is 1

# Create grid to evaluate model
xy, xx, yy = svm_def.create_grid(x, 10)

""" Tune model """
# Set parameters to be tuned. Other options can be added
tune_parameters = {'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]}

# Tune SVM
clf_tune = GridSearchCV(svm.SVC(kernel='linear'), tune_parameters)
clf_tune.fit(x, y)
```

```
[67]: GridSearchCV(estimator=SVC(kernel='linear'),
                   param_grid={'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]})
```

We can easily access the cross-validation errors for each of these models using the **best__params__** and **cv__results__** attributes:

```
[68]: print("Best Parameters:\n", clf_tune.best_params_)
      print("Mean scores:\n", clf_tune.cv_results_['mean_test_score'],
            "\nStandard Deviation scores:\n", clf_tune.cv_results_['std_test_score'],
            "\nRanking scores:\n", clf_tune.cv_results_['rank_test_score'])
```

```
Best Parameters:
 {'C': 0.1}
Mean scores:
 [0.65 0.65 0.7  0.55 0.55 0.55 0.55]
Standard Deviation scores:
 [0.2        0.2        0.18708287 0.18708287 0.18708287 0.18708287
 0.18708287]
Ranking scores:
 [2 2 1 4 4 4 4]
```

We see that **cost=0.1** results in the lowest cross-validation error rate. The optimal model can be fitted using the **best__params__[‘C’]** for the cost variable.

[69]:
```
# Optimal parameter
clf_opt = svm.SVC(kernel='linear', C=clf_tune.best_params_['C'])
clf_opt.fit(x, y)
Z_opt = clf_opt.decision_function(xy).reshape(xx.shape)
```

The function **predict()** can be used to predict the class label on a set of test observations, at any given value of the cost parameter. We begin by generating a test data set.

[70]:
```
# Generate test dataset
np.random.seed(4)
# x from normal distribution # random y
xtest = np.random.normal(loc = 2, scale = 1.0, size = (2*n, 2))
ytest = np.random.choice((-1, 1), 2*n)

xtest[ytest==1, :] += 1     # offset x[:][:] for y is 1
```

Now we predict the class labels of these test observations. Here we use the best model obtained through cross-validation in order to make predictions.

[71]:
```
# Prediction
ypred_opt = clf_opt.predict(xtest)

# Create table
tab_opt = svm_def.table_scores(ypred_opt, ytest)

print("Optimal C = ", clf_tune.best_params_['C'], " scores:\n", tab_opt)
```

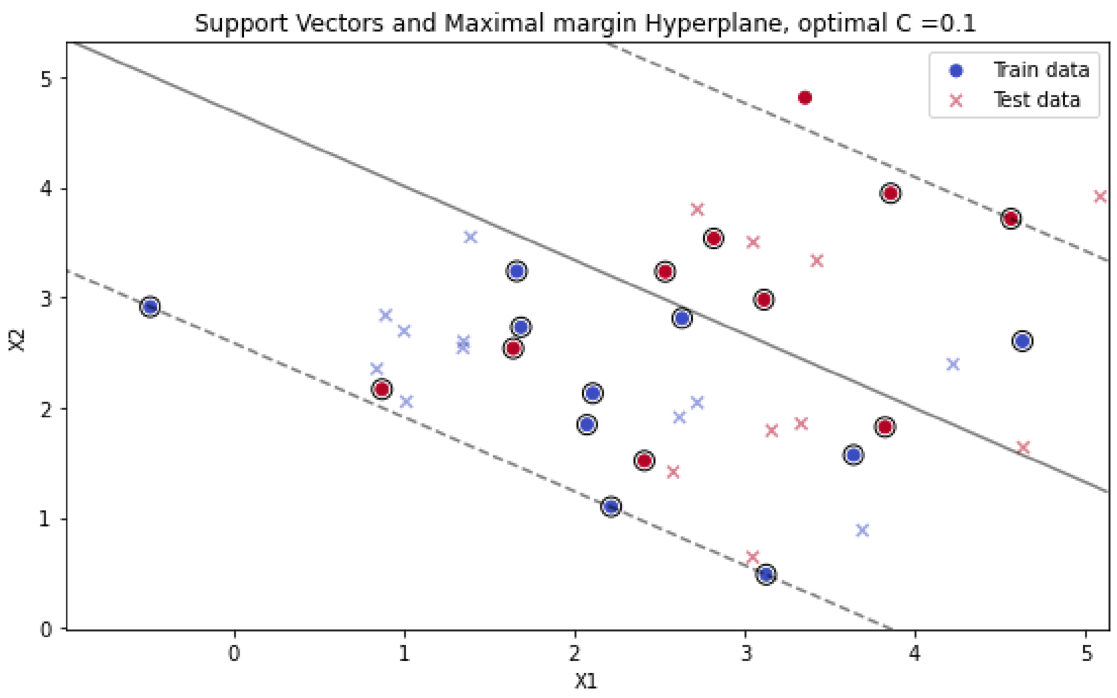
```
Optimal C =  0.1  scores:
          Pred -1  Pred 1
True -1     10.0     1.0
True 1       4.0     5.0
```

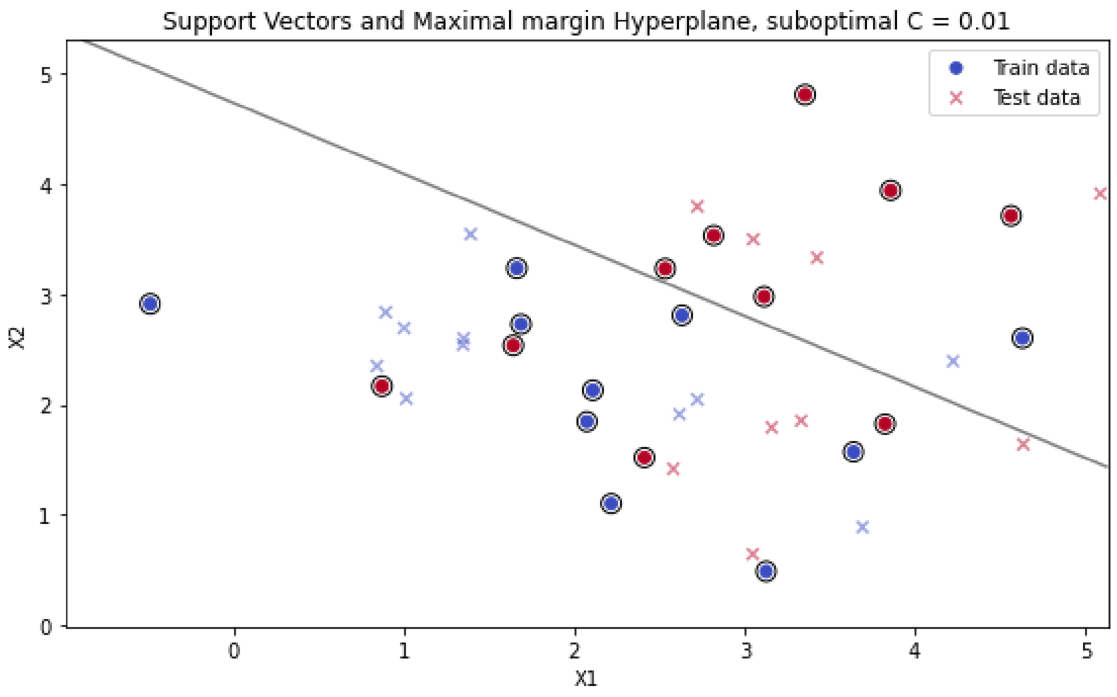
Thus, with this value of **cost**, 15 of the test observations are correctly classified. What if we had instead used **cost=0.01**?

[72]:
```
# Suboptimal parameter c = 0.01
clf_001 = svm.SVC(kernel='linear', C=0.01)
clf_001.fit(x, y)
Z_001 = clf_001.decision_function(xy).reshape(xx.shape)

# Prediction
ypred_001 = clf_001.predict(xtest)

# Create table
tab_001 = svm_def.table_scores(ypred_001, ytest)

print("\nAnother C = 0.01 scores:\n", tab_001)
```

```
Another C = 0.01 scores:
          Pred -1  Pred 1
True -1     10.0     1.0
True 1       5.0     4.0
```

[73]:
```
""" Plots """
fig = plt.figure(figsize=[8, 10])

# Create scatter plot data
ax = fig.add_subplot(2, 1, 1)
svm_def.svm_plot(ax, x, y, Z_opt, clf_opt, xtest=xtest,
                 ytest=ytest, plottest=True)

plt.legend(["Train data", "Test data"])
title = ("Support Vectors and Maximal margin Hyperplane, optimal C ="
         + str(clf_tune.best_params_['C']))
plt.title(title)

# Plot alternative fit, scatter plot data
ax = fig.add_subplot(2, 1, 2)
svm_def.svm_plot(ax, x, y, Z_001, clf_001, xtest=xtest,
                 ytest=ytest, plottest=True)
plt.legend(["Train data", "Test data"])
plt.title("Support Vectors and Maximal margin Hyperplane, suboptimal C = 0.01")

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





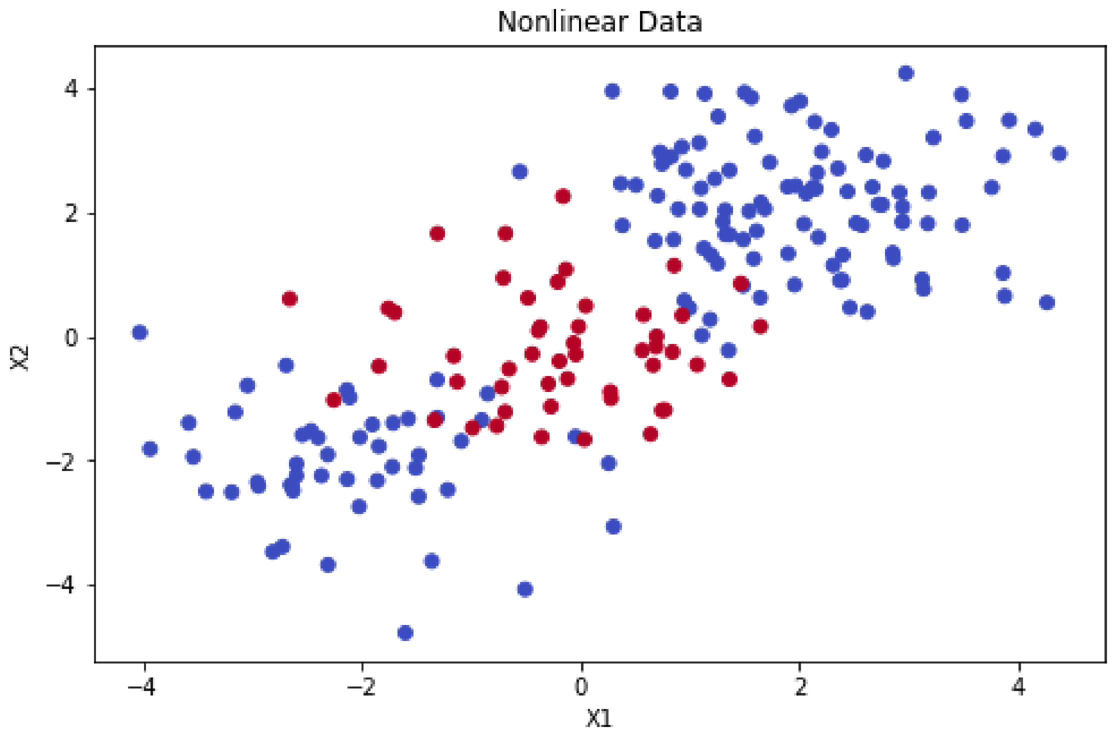
6.4.3 ROC Curve

- the ROC curve is a popular graphic for simultaneously displaying the two types of errors - **false positive** and **true positive** rates for all possible thresholds
- ROC curve is constructed by:

- Choosing threshold value
- Computing fitted values $f(x^*)$ of observations
- Classifying observations with respect to the chosen threshold
- Computing the related true and false positive rates
- For each threshold, display the corresponding true and false positive rate as ROC curve

ROC Curve code example

[74]:
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from sklearn import svm
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import plot_roc_curve
from SVM_def import SVM_def

# SVM_def is a class containing definitions used throughout this Chapter
svm_def = SVM_def()

# Create nonlinear data
n = 100
np.random.seed(0)
# x from normal distribution # y is 1 or 2
x = np.random.normal(scale = 1.0, size = (2*n, 2))
y = np.concatenate((np.ones(int(3/2*n)), 2*np.ones(int(1/2*n))))

x[:n, :] += 2
x[n:int(n*1.5), :] -= 2

#Plot
fig = plt.figure(figsize=[8, 5])

# Create scatter plot data
ax = fig.add_subplot(1, 1, 1)
ax.scatter(x[:,0], x[:,1], c=y, cmap=cm.coolwarm,  marker="o")

plt.xlabel("X1"), plt.ylabel("X2")
plt.title("Nonlinear Data")
plt.show()
```

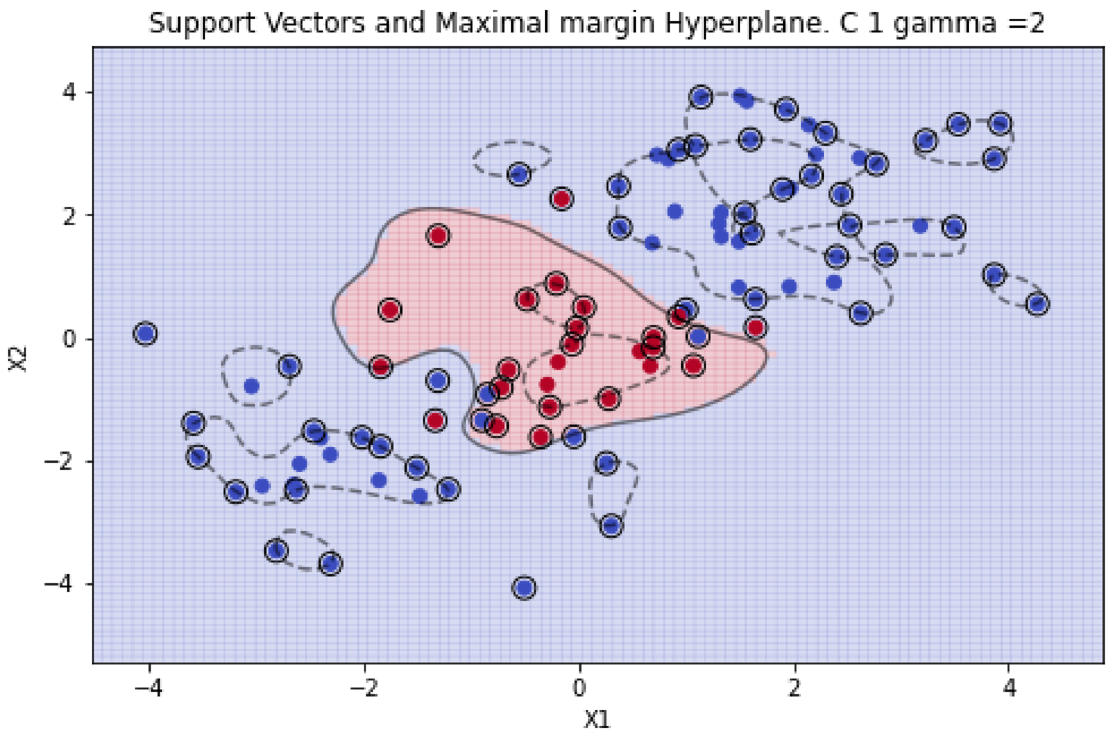


[75]:
```
""" First fit Radial Kernel """
# Create grid to evaluate model
xy, xx, yy = svm_def.create_grid(x, 100)

# divide training data
np.random.seed(0)
itrain = np.random.choice(200, 100, replace=False)
# Test dataset
xtest = np.delete(x, itrain, axis=0)
ytest = np.delete(y, itrain, axis=0)

c, gamma = 1, 2
clf_g2 = svm.SVC(kernel='rbf', C=c, gamma=gamma)
clf_g2.fit(x[itrain], y[itrain])
Z_g2 = clf_g2.decision_function(xy).reshape(xx.shape)
```

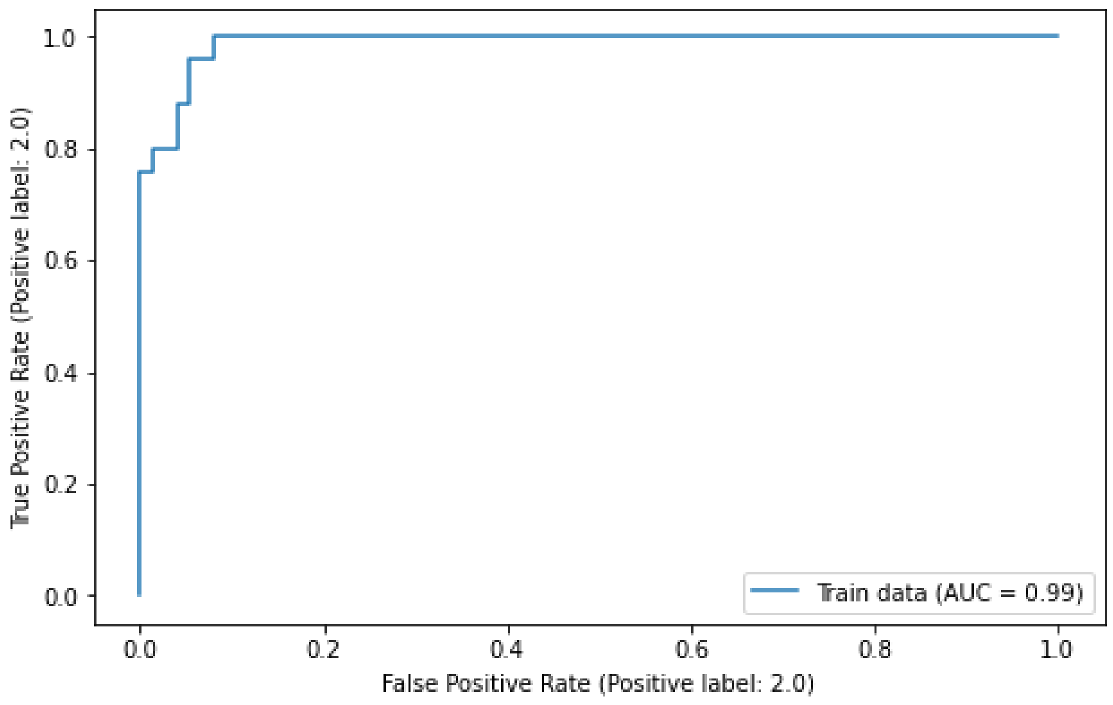
[76]:
```
## plot model
fig = plt.figure(figsize=[8, 5])
ax = fig.add_subplot(1, 1, 1)
svm_def.svm_plot(ax, x[itrain, :], y[itrain], Z_g2, clf_g2, coloring=True)

title = ("Support Vectors and Maximal margin Hyperplane. C " + str(c)
         + " gamma =" + str(gamma))
```

```
plt.title(title)

print("Number of Support Vectors: ", len(clf_g2.support_))
```

```
Number of Support Vectors:  70
```



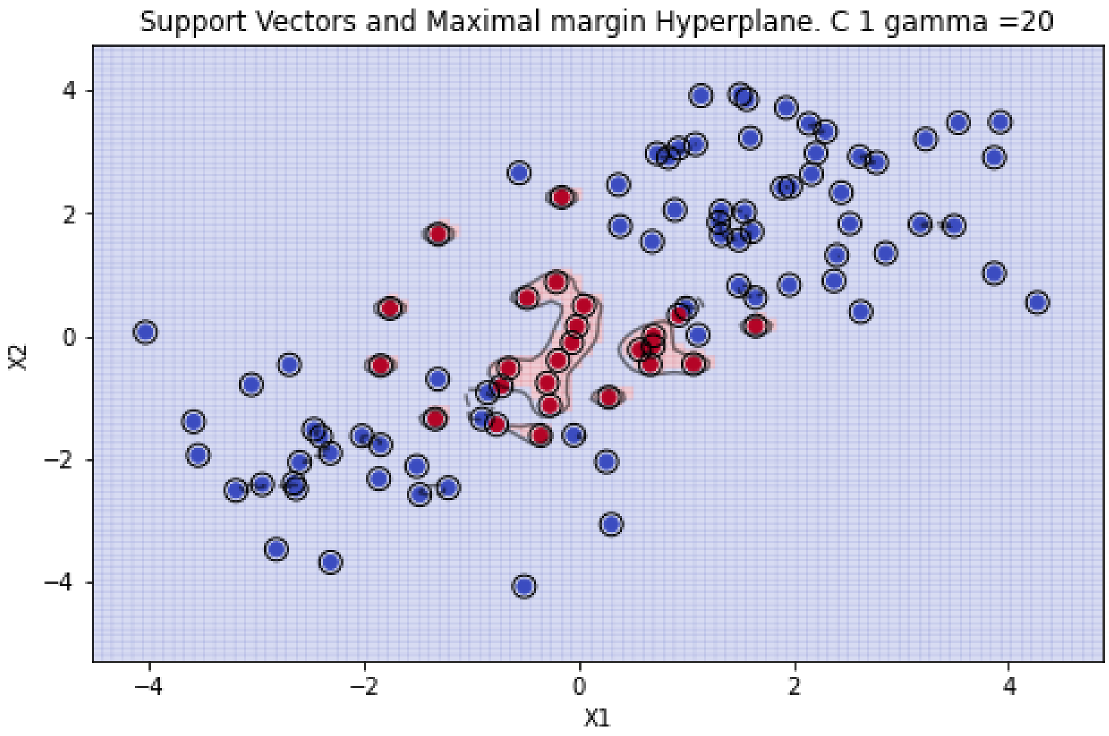
[77]:
```
""" ROC plot using plot_roc_curve """
fig = plt.figure(figsize=[8, 5])
ax = fig.add_subplot(1, 1, 1)
# plot_roc_curve(clf_g2, xtest, ytest, ax=ax, name="Test Data")
plot_roc_curve(clf_g2, x[itrain], y[itrain], ax=ax, name="Train data")
plt.show()
```

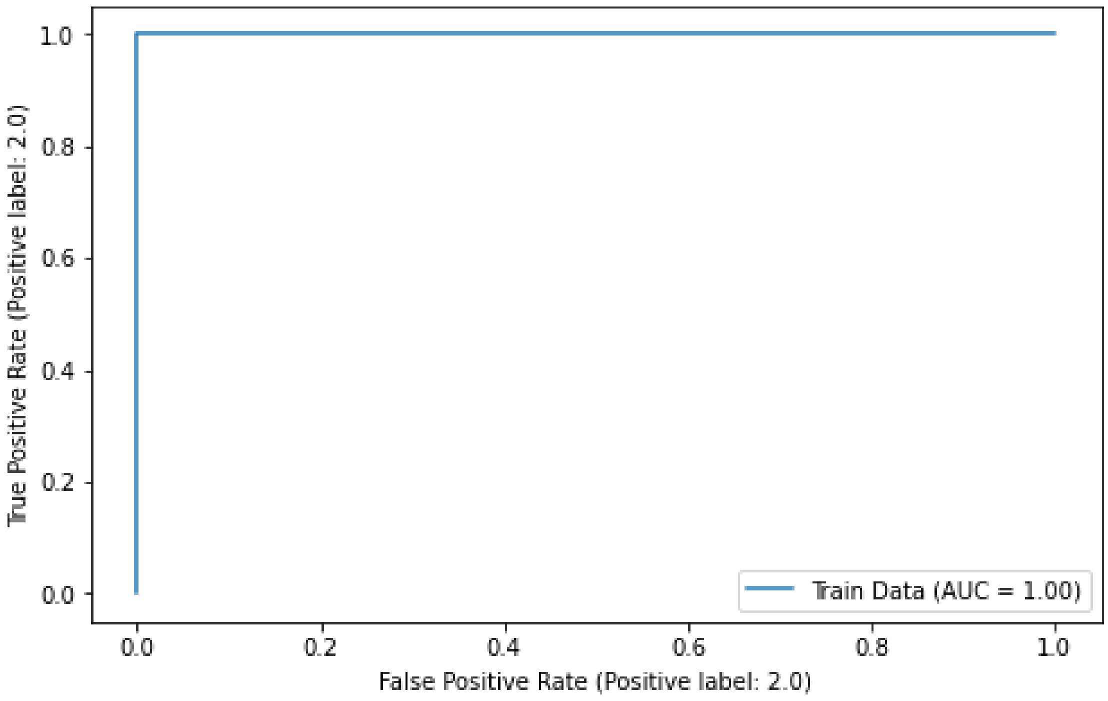


[78]:
```
""" Increased gamma in fit  """
# New model
c, gamma = 1, 20
clf_g20 = svm.SVC(kernel='rbf', C=c, gamma=gamma)
clf_g20.fit(x[itrain], y[itrain])
Z_g20 = clf_g20.decision_function(xy).reshape(xx.shape)

# plot model
fig = plt.figure(figsize=[8, 5])
ax = fig.add_subplot(1, 1, 1)
svm_def.svm_plot(ax, x[itrain, :], y[itrain], Z_g20, clf_g20,
                 coloring=True)
title = ("Support Vectors and Maximal margin Hyperplane. C " + str(c)
         + " gamma =" + str(gamma))
plt.title(title)

print("Number of Support Vectors: ", len(clf_g20.support_))

""" ROC plot using plot_roc_curve: Train data flexible """
fig = plt.figure(figsize=[8, 5])
ax = fig.add_subplot(1, 1, 1)
# plot_roc_curve(clf_g20, xtest, ytest, ax=ax, name="Test Data")
plot_roc_curve(clf_g20, x[itrain], y[itrain], ax=ax, name="Train Data")
```

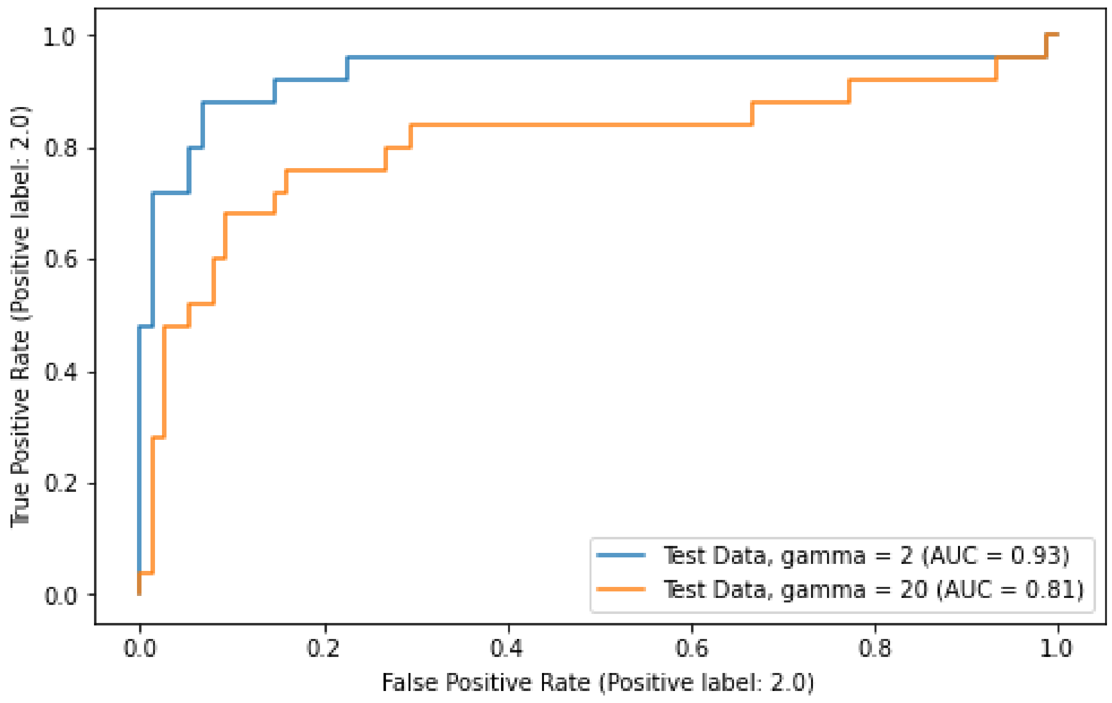
```
plt.show()
```

```
Number of Support Vectors:  100
```





[79]:
```
""" ROC plot using plot_roc_curve: Test data """
fig = plt.figure(figsize=[8, 5])
ax = fig.add_subplot(1, 1, 1)
plot_roc_curve(clf_g2, xtest, ytest, ax=ax, name="Test Data, gamma = 2")
plot_roc_curve(clf_g20, xtest, ytest, ax=ax, name="Test Data, gamma = 20")

plt.show()
```



6.4.4 SVM with more than 2 classes

- One-versus-One approach
- One-versus-all approach

One-versus-One Classification

- A one-versus-one or all-pairs approach constructs $\frac{K}{2}$ SVMs, each of which compares a pair of classes
- **Example**: one such SVM might compare the kth class, coded as +1, to the k'th class, coded as —1
- We classify a test observation using each of the $\frac{K}{2}$ classifiers, and we tally the number of times that the test observation is assigned to each of the K classes

- The final classification is performed by assigning the test observation to the class to which it was most frequently assigned in these $\frac{K}{2}$ pairwise classifications (**majority vote** essentialy)

One-versus-All classification

- We fit K SVMs, each time comparing one of all the K classes to the remaining K—1 classes
- Let $\beta_0 k, \alpha_1 k, ..., \alpha_p k$ denote the parameters that result from fitting an SVM comparing the kth class (coded as +1) to the others (coded as —1)
- Let x^* denote a test observation. We assign the observation to the class for which

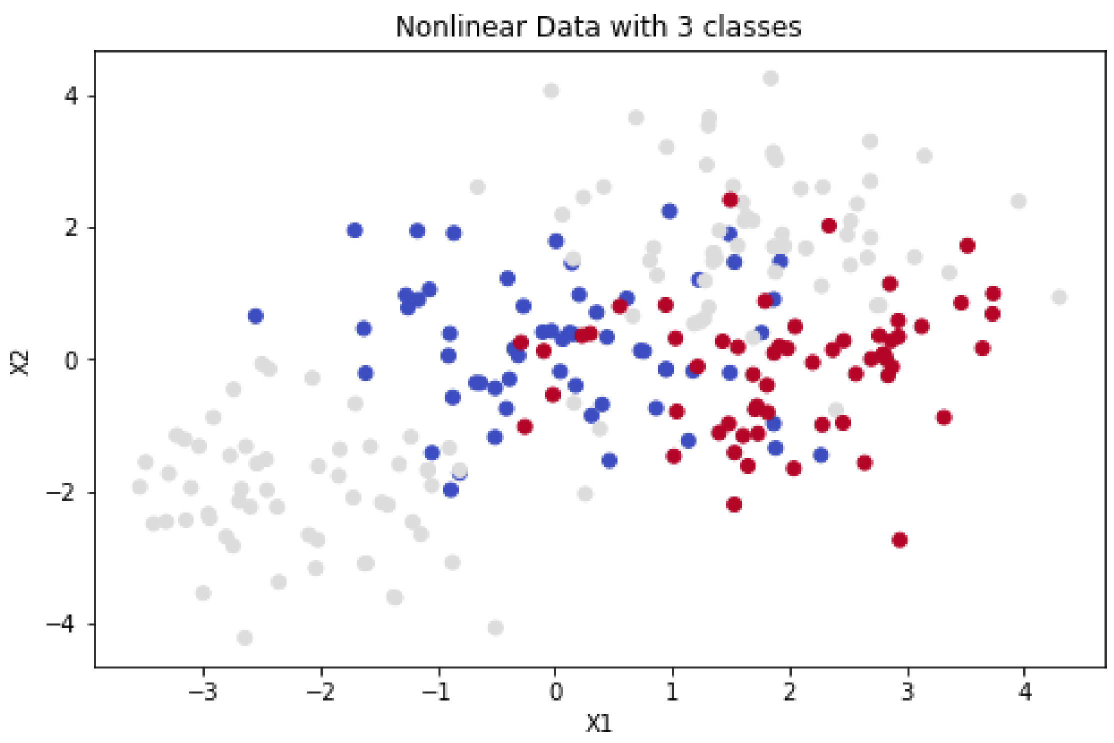
$$f(x^*) = \beta_0 k + \sum_{i \in \mathcal{S}} \alpha_{ik} K(x^*, x_i)$$

  is **largest**
- This amounts to a high level of confidence that the test observation belongs to the kth class rather than to any of the other classes.

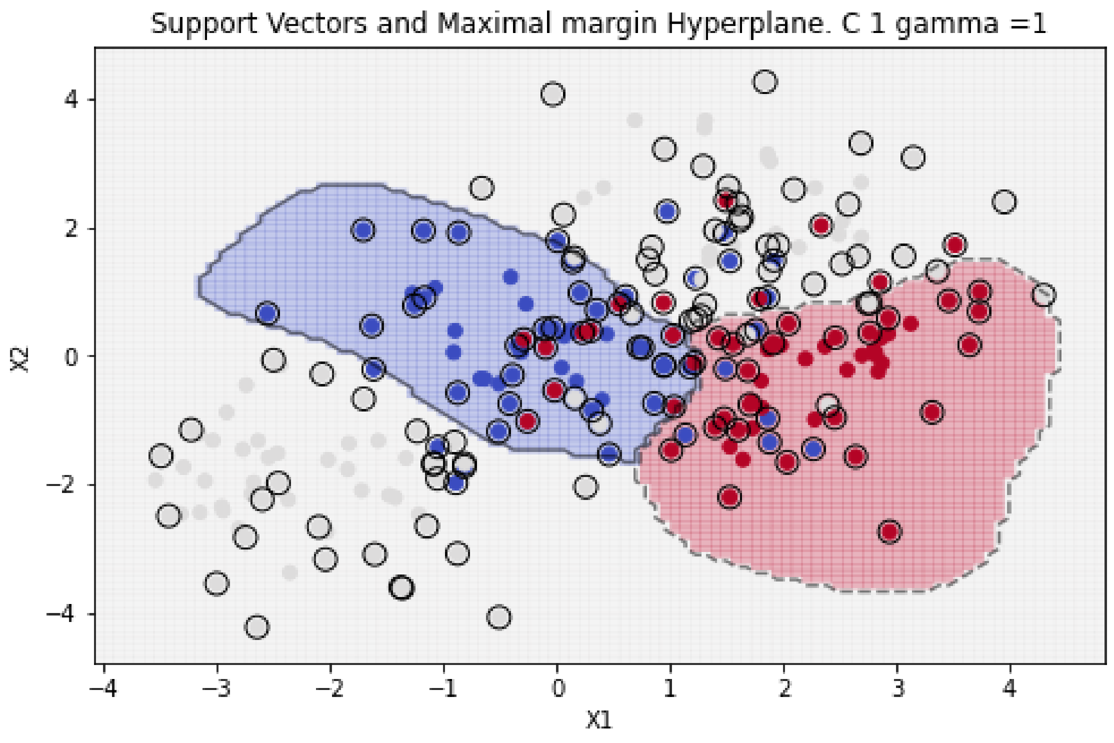
Code Example One-versus-One and One-versus-All

[80]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from sklearn import svm
from SVM_def import SVM_def

# SVM_def is a class containing definitions used throughout this Chapter
svm_def = SVM_def()

# Create nonlinear data
n = 120
np.random.seed(0)
# x from normal distribution # y is 0, 1, 2
x = np.random.normal(scale = 1.0, size = (2*n, 2))
y = np.concatenate((np.zeros(int(1/2*n)),
                    np.ones(int(2/2*n)), 2*np.ones(int(1/2*n))))

x[:int(n/2), :] += 0
x[int(n/2):int(n*2/2), :] -= 2
x[int(n*2/2):int(n*3/2), :] += 2
x[int(n*3/2):, 0] += 2

fig = plt.figure(figsize=[8, 5])

# Create scatter plot data
ax = fig.add_subplot(1, 1, 1)
ax.scatter(x[:,0], x[:,1], c=y, cmap=cm.coolwarm,  marker="o")

plt.xlabel("X1"), plt.ylabel("X2")
plt.title("Nonlinear Data with 3 classes")
plt.show()
```

```
""" Fit radial kernel on 3 classes """
# Create grid to evaluate model
xy, xx, yy = svm_def.create_grid(x, 100)

c, gamma = 1, 1
clf = svm.SVC(kernel='rbf', C=c, gamma=gamma, probability=True)
clf.fit(x, y)
Z = clf.predict(xy).reshape(xx.shape)

# plot model
fig = plt.figure(figsize=[8, 5])
ax = fig.add_subplot(1, 1, 1)

plt.pcolor(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.3,
           edgecolors='face')
#shading='auto'

svm_def.svm_plot(ax, x, y, Z, clf)

title = ("Support Vectors and Maximal margin Hyperplane. C " + str(c)
         + " gamma =" + str(gamma))
plt.title(title)

print("Number of Support Vectors: ", len(clf.support_))

plt.show()
```



```
Number of Support Vectors:  148
```



7 Decision Trees

- for **classification** and **regression**
- aims at **segmenting/stratifying** the predictor space into regions
- each region is equivalent to a class assignment
- **Pro**
  - simple
  - interpretable (by visual representation for example)
  - can work with categorical and numerical feature alike
  - Requires little data preprocessing: no need for one-hot encoding, dummy variables, and so on.
  - Non-parametric model: no assumptions about the shape of data.
  - Fast for inference.
  - Feature selection happens automatically: unimportant features will not influence the result. The presence of features that depend on each other (multicollinearity) also doesn't affect the quality.
- **Con**
  - typically not competitive with the best supervised learning methods by predictive accuracy
  - It tends to overfit
  - depending on data size, long computation
  - **Decision tree for many features**: Take more time for training-time complexity to increase as the input increases.

7.1 Binary Splitting

- Idea of a decision tree is to recursively partition the predictor variable space into disjoint regions, where each region is as pure as possible with respect to the labels of the response variable
- These regions are referred to as **terminal nodes** or **leaves**.
- Assume we are given a training set $(\vec{x}_1, y_1), ..., (\vec{x}_n, y_n)$ where $\vec{x}_i = x_{i1}, ..., x_{ip}$ are predictor values and y_i is a qualitative response (not necessairly binary)
  1. Init the set of regions $\mathcal{R} = \{R\}$ by the predictor domain R (e.g. $\mathbb{R}^p$)
  2. Choose the optimal region R in $\mathcal{R}$ and the optimal predictoe X_i such that a **binary split** of R with respect to X_i

     $$R_1 = \{x \in \mathbb{R} | x_i > t\} \text{ and } R_2 = \{x \in \mathbb{R} | x_i \leq t\}$$

     given the *highest gain* in purity (for some threshold t)
  3. Replace R in $\mathbb{R}$ by R_1 and R_2 and return to 2
- iteration is stopped when a stopping criterion is fullfilled
  - gain in purity
  - depth

7.2 Node Purity

- refers to a quantitative measure of how the most frequent occuring class in a region is dominant (e.g. 90% class 3 here)

7.2.1 Classification Error Rate

- is the fraction of training samples in a region that do not belong to the most common class

$$E_m(T) = 1 - \max_k \hat{p}_{mk}$$

  - $\hat{p}_{mk}$ is the proportion of the training data in redion m that is from level k
  - **but** classification error is not sufficiently sensitive to tree-growing

7.2.2 Gini Index

- measure of total variance across K classes:

$$G_m(T) = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

  - takes on a small value if all the p_{mk} are close to zero or 1
  - is referred to as node **purity** measure: a small value indicates that a node contains predominantly observations of the same class

7.2.3 Cross-entropy

- alternative to Gini-Index

$$D_m(T) = -\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk})$$

- similar to Gini-Index numerically

7.3 Pruning a Tree

- growing a complex tree on the training data will likely overfit on the test data
- smaller tree with fewer splits (with fewer regions $R_1, R_2, ..., R_j$) might lead to lower variance and better interpretation at the cost of a little bias
- overcoming this by growing the tree as long as the purity measure at each split exceeds some (high) threshold
  - however, doing only this is short sighted since a worthless split could be followed by a very good split
- **Tree pruning** therefore grows a very large tree and successively **prune** it back until good **subtree**

7.3.1 Cost Complexity Pruning

- We first compute a very large tree T_0

- We consider a sequence of subtrees T_α indexed by a nonnegative tuning parameter α : for each value of α there corresponds a subtree $T \subset T_0$ which minimizes the **cost-complexity function**:

$$R_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|, \quad \alpha \geq 0$$

  where

  - $|T|$ denotes the number of terminal nodes in the tree T
  - m ranges over all terminal nodes
  - For each terminal node m, the number N_m denotes the number of elements in the node
  - $Q_m(T)$ is an arbitrary node purity measure (e.g. the Gini index $G_m(t)$)

- For each given value of α we compute a T_α such that $R_\alpha(T)$ is minimized

- Here α is a tuning parameter that controls the trade-off between complexity and fit to the training data

- A large value of α penalizes the number of terminal nodes of the tree and hence the resulting tree T will have few terminal nodes, but potentially will fit poorly to the training data

- Vice versa, a small value of α puts all emphasis on data fit and a complex tree with many terminal nodes will result. In the extreme case that $\alpha = 0$, we will have $T = T_0$.

- Finding optimal

Cost complexity pruning 1. Use recursive binary splitting to grow a very large tree T_0 and set $\alpha = 0$ 2. Increase and for each value compute the subtree T_α that minimizes the cost-complexity function $R_\alpha(T)$ 3. Stop if T_α is root node

Optimal pruning by cross-validation

- Devide dataset into K folds. For $k = 1, ..., K$
  1. Grow a large tree T_0 based on all but the k-th fold of the data set
  2. Perform cost-complexity pruning and obtain a sequence of optimal subtrees T_α as a function of . Compute the classification error rate on the k-th fold as a function of
- for each value of average the classification error rate of the K folds.
- Choose such that the error rate is minimal

7.4 Decision Tree versus Logistic Regression

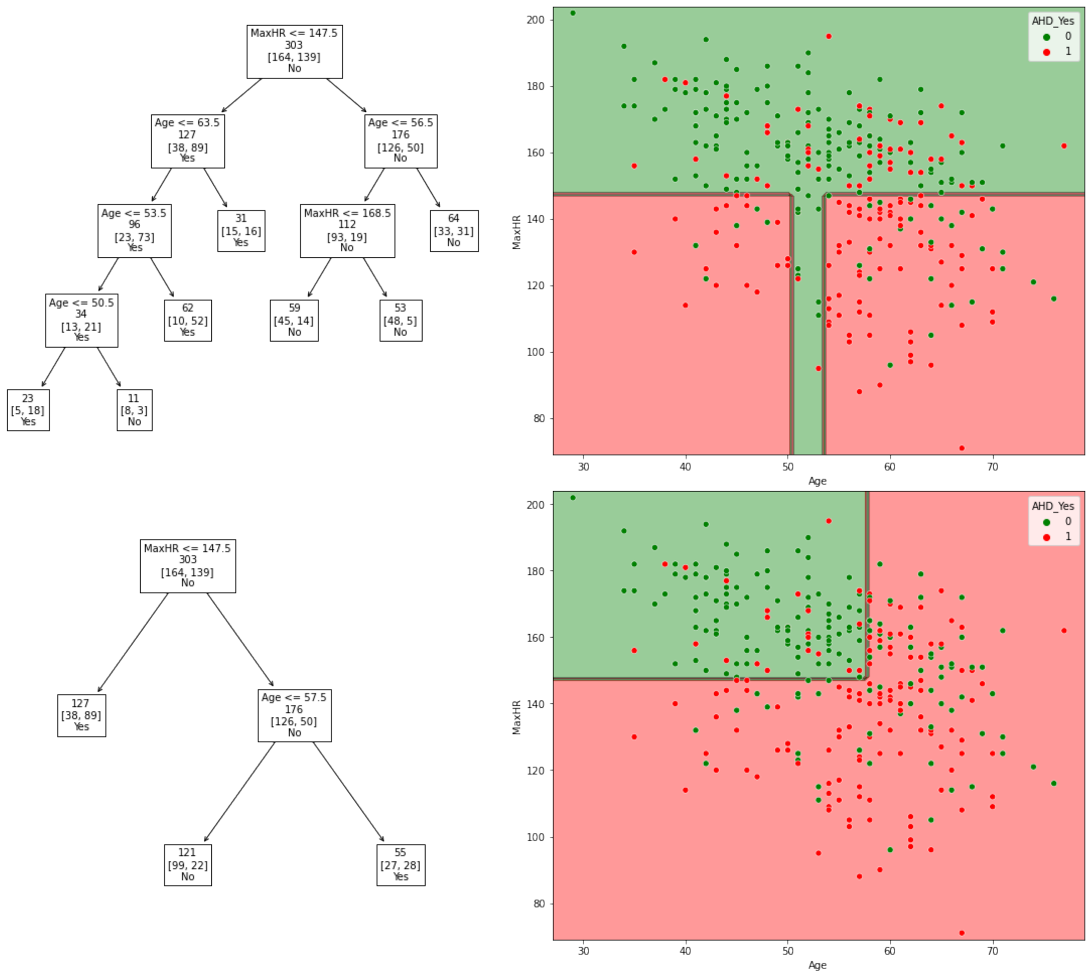
- logistic regression partitions the predictor space in two half spaces by a hyperplane
- decision trees partition feature space into axis parallel boxes

7.5 Code Example of Decision Tree with different criteria and confusion matrix

[81]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import tree
```

```python
# Load data
df = pd.read_csv(path_data+'Heart.csv')

# Replace Categorical Variable with dummies
df = pd.get_dummies(data=df, columns=['AHD'], drop_first=True)

# Define x and y
y = df[['AHD_Yes']]
X = df[['Age', 'MaxHR']]

# Create grid
margin, n = 2, 100
xx = np.linspace(min(X['Age']) - margin, max(X['Age']) + margin, n)
yy = np.linspace(min(X['MaxHR']) - margin, max(X['MaxHR']) + margin, n)
yy, xx = np.meshgrid(yy, xx)

# Plots
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(16, 14))

# Classifier settings:
crit = ['entropy', 'gini']
mss = [10, 60]
# For each classifier criterion:
for i in range(len(crit)):
    # Create and fit Decision tree classifier
    clf = tree.DecisionTreeClassifier(criterion=crit[i],
                                      min_samples_split=mss[i],
                                      min_samples_leaf=5,
                                      min_impurity_decrease=0.01)
    clf = clf.fit(X, y)
    # Predict
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot surface
    ax[i][1].contourf(xx, yy, Z, alpha=0.4,
                      colors=['green', 'black', 'red', 'black'])

    # Plot datapoints
    sns.scatterplot(x=df['Age'], y=df['MaxHR'],  ax=ax[i][1],
                    hue=df['AHD_Yes'], palette=['green', 'red'])

    # Plot Decision Tree
    tree.plot_tree(clf, ax=ax[i][0], fontsize=10, impurity=False,
                   label='Root', feature_names=X.columns.values,
                   class_names=['No', 'Yes'])
```

```
plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)
plt.show()
```



[82]:
```
import numpy as np
import pandas as pd
from sklearn import tree

# Load data
df = pd.read_csv(path_data+'Heart.csv')

# Replace Categorical Variable with dummies
df = pd.get_dummies(data=df, columns=['AHD'], drop_first=True)
df['ChestPain'], ChestPain_codes = pd.factorize(df['ChestPain'])
df['Thal'], Thal_codes = pd.factorize(df['Thal'])
# Drop NA rows:
df.dropna(inplace=True)
df.reset_index(inplace=True) # After removing NA
```

```python
# Split in test-train
np.random.seed(0)
i = df.index
# Index of train
i_train = np.random.choice(i, replace=False, size=int(250))

# Save DataFrames
df_train = df.iloc[i_train]
df_test = df.drop(i_train)

# Define x and y
y_train = df_train['AHD_Yes']
y_test = df_test['AHD_Yes']
X_train = df_train.drop(columns=['AHD_Yes'])
X_test = df_test.drop(columns=['AHD_Yes'])

# Create and fit Decision tree classifier
clf = tree.DecisionTreeClassifier(criterion='entropy',
                                  min_samples_split=2,
                                  min_samples_leaf=1,
                                  min_impurity_decrease=0.0001)
clf = clf.fit(X_train, y_train)

# Predictions:
y_train_pred = clf.predict(X_train)
y_test_pred = clf.predict(X_test)

# Create confusion matrix
def confusion(y_true, y_pred):
    conf = pd.DataFrame({'predicted': y_pred, 'true': y_true})
    conf = pd.crosstab(conf.predicted, conf.true,
                       margins=True, margins_name="Sum")
    return conf

print('Test data:\n',
      confusion(y_test.T.to_numpy(), y_test_pred))
print('\n\nTrain data:\n',
      confusion(y_train.T.to_numpy(), y_train_pred))

# Classification error:
err_test = abs(y_test - y_test_pred).mean()
err_train = abs(y_train - y_train_pred).mean()

print('\n')
print('Classification error on Testdata:\n', np.round(err_test, 3),
     '\nClassification error on Traindata:\n', np.round(err_train, 3))
```

```
Test data:
 true        0   1  Sum
predicted
0           25   6   31
1            5  13   18
Sum         30  19   49


Train data:
 true         0    1  Sum
predicted
0           131    0  131
1             0  119  119
Sum         131  119  250


Classification error on Testdata:
 26.143
Classification error on Traindata:
 0.0
```

7.6 Code Example Decision Tree with cost complexity pruning and cross validation

We perform cost-complexity pruning to the **heart** data set. The function **cv.tree** in the **tree**-package provides the implementation of cross-validation for choosing an optimal value of α. The function returns a list that contains the vectors

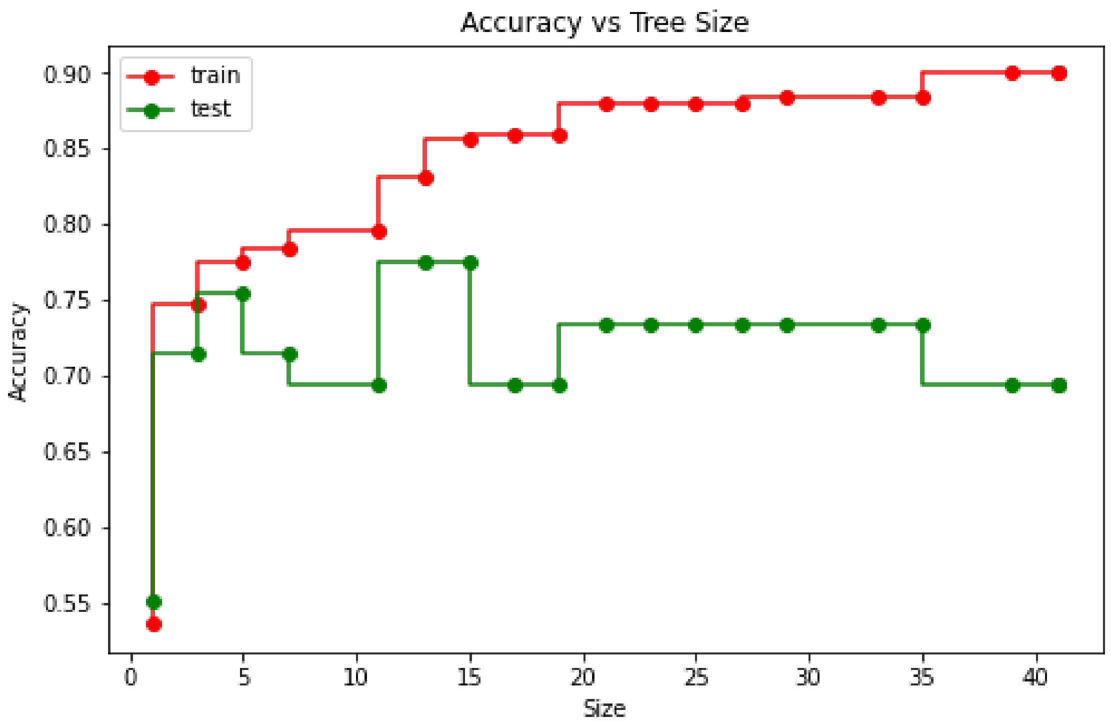
- **path** is the vector of those α values, where changes in T_α occur
- **score** contains the corresponding averaged classification accuracies
- **node** is the vector containing the corresponding tree sizes.

The following code computes a large tree and performs cost-complexity pruning where the parameter α is chosen by 10-fold cross-validation.

[83]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import tree

# Load data
df = pd.read_csv(path_data+'Heart.csv')

# Replace Categorical Variable with dummies
df = pd.get_dummies(data=df, columns=['AHD'], drop_first=True)
df['ChestPain'], ChestPain_codes = pd.factorize(df['ChestPain'])
df['Thal'], Thal_codes = pd.factorize(df['Thal'])
# Drop NA rows:
df.dropna(inplace=True)
df.reset_index(inplace=True) # After removing NA

# Split in test-train
np.random.seed(2)
i = df.index
# Index of train
i_train = np.random.choice(i, replace=False, size=int(250))

# Save DataFrames
df_train = df.iloc[i_train]
df_test = df.drop(i_train)

# Define x and y
y_train = df_train['AHD_Yes']
y_test = df_test['AHD_Yes']
X_train = df_train.drop(columns=['AHD_Yes'])
X_test = df_test.drop(columns=['AHD_Yes'])

# Create and fit Decision tree classifier
clf = tree.DecisionTreeClassifier(criterion='entropy',
                                  min_samples_split=10,
                                  min_samples_leaf=5,
                                  min_impurity_decrease=0.005)

path = clf.cost_complexity_pruning_path(X_train, y_train)
```

We finally compare the classification error rate of the training set with the cross-validated error. Below these two errors are plotted against the number of nodes in the tree (tree size). It becomes obvious that the training error shrinks to zero as the complexity of the tree increases whereas the cross-validated value has a minimum for the tree above with 12 terminal nodes.

[84]:
```
# Find misclasses and size of respective Trees T(alpha)
# Train:
node = []
score_train, score_test = [], []

for alpha in path.ccp_alphas:
    # Create and fit Decision tree classifier
    clf = tree.DecisionTreeClassifier(criterion='entropy',
                                      min_samples_split=10,
                                      min_samples_leaf=5,
                                      min_impurity_decrease=0.005,
                                      ccp_alpha=alpha)
    clf = clf.fit(X_train, y_train)
    # Save node count:
    node.append(clf.tree_.node_count)
    # Save Scores
    score_train.append(clf.score(X_train, y_train))
    score_test.append(clf.score(X_test, y_test))

# Plot Score vs Size
fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(node, score_train,
        'r-o', drawstyle="steps-post", label='train')
ax.plot(node, score_test,
        'g-o', drawstyle="steps-post", label='test')
ax.set_xlabel("Size")
ax.set_ylabel("Accuracy")
ax.set_title("Accuracy vs Tree Size")
plt.legend()
plt.show()
```



8 Bagging, Bootstrapping and Random Forests

8.1 Bagging

- allows for reducing the variance of a statistical estimator (e.g. Decision Tree)
- is based on the **aggregation** of a multitude of estimators with high variance by means of averaging and thus reducing variance
- estimators are created by **bootstrapping** the training data (bagging = **b**ootstrap

aggregat**ing**)
- Random Forest incorporates this method

8.2 Bootstrapping

- Rather than repeatedly obtaining independent data sets from the population, we instead obtain distinct data sets by repeatedly sampling observations from the original data set with replacement
- Each of these bootstrap data sets is created by sampling with replacement and is the same size as our original data set. As a result, some observations may appear more than once in a given bootstrap data set and some not at all
- in other words: take a measurement out of a pool. throw it back, shake it, take the next one out

Bootstrapping

Let $x_1, \ldots, x_n$ be (possibly multivariate) realizations of independent and identically distributed random variables $X_1, \ldots, X_n$. Assume further that

$$\hat{\gamma} = \hat{\gamma}\left(x_1, \ldots, x_n\right)$$

is an estimator of some quantity γ. (an estimator, can be mean, variance, model etc.) 1. Choose a (large) number $B \in \mathbb{N}$. 2. For $b = 1, \ldots, B$ * Draw n samples $\{x_1^*, \ldots, x_n^*\}$ from $\{x_1, \ldots, x_n\}$ with replacement. * Compute the estimator $\hat{\gamma}_b^* = \hat{\gamma}\left(x_1^*, \ldots, x_n^*\right)$ 3. The emperical distribution funtion $\hat{F}^*\ of\ (\hat{\gamma}_1^*, ..., \hat{\gamma}_B^*)$ approximates the distribution of $\hat{\gamma}$. In particular, the standard error of $\hat{\gamma}$ can be estimated by the bootstrap estimate

$$\mathrm{se}_B(\hat{\gamma}) = \sqrt{\frac{1}{B-1}\sum_{b=1}^{B}\left(\hat{\gamma}_n^* - \bar{\gamma}^*\right)^2}, \quad \text{with}$$
$$\bar{\gamma}^* = \frac{1}{B}\sum_{b=1}^{B}\hat{\gamma}_b^*.$$

* note: each individual bootstrap sample $x_1^*, ..., x_n^*$ will contain duplicates of values in the original data set and likewise some values will not appear at all

8.3 Bagging - Bootstrap aggregation

- Bootstrap aggregation, or bagging : is a general-purpose procedure for reducing the variance of a statistical learning method; it is particularly useful and frequent in the context of decision trees
- Recall that for a sample of independent random variables $Z_l, ..., Z_n$ each with variance σ^2 , the variance of the mean $\overline{Z}$ is only $\frac{\sigma^2}{n}$
  - if you take the whole population, the variance is O since you have the whole with only 20 samples, variance is quite large
- By **averaging** we reduce the variance
- Natural way to decrease the variance of a statistical learning method and hence to increase the predictive accuracy would consist of
  - taking many training sets from the population
  - building a separate prediction model using each training set

    - averaging the resulting predictions
- This procedure is not practical since we do not have access to several training sets
- Bootstrap by taking repeated samples from (single) training set
- In other words, we generate B different bootstrapped training sets and for each of these sets we compute a predictive model $\hat{f}_1^*(x), ..., \hat{f}_B^*(x)$

Long story short: fake a whole ass of training sets by resampling and replacement and train your shit with it

8.3.1 Bagging in Regression

- If we are in the regression setting (imagining that each of the $\hat{f}_b^*$ is a linear regression model) then the bagged model is simply the average

$$\hat{f}_b ag(x) = \frac{1}{B}\sum_{i=1}^{B} \hat{f}_i^*(x)$$

8.3.2 Bagging in Classification

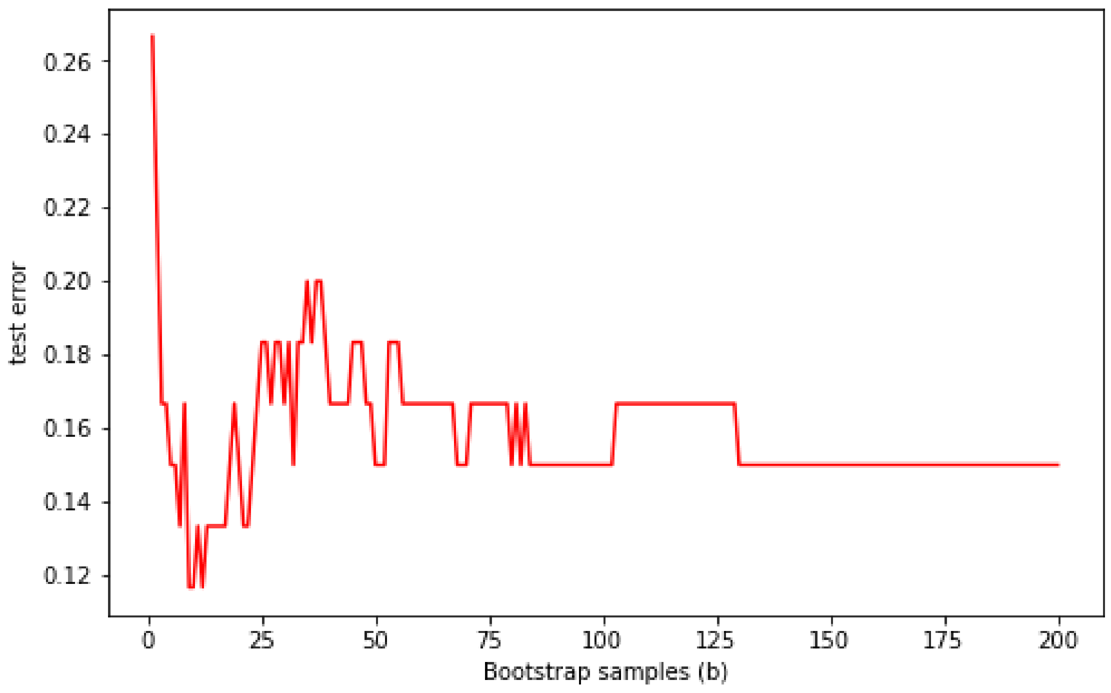
- For classification typically the predicted classes of all B models are recorded and the most commonly occurring class among the B predictions is used.
- **majority vote**

8.3.3 Code Example Decision Tree with Bagging

[85]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

# Load data
df = pd.read_csv(path_data+'Heart.csv')

# Replace Categorical Variable with dummies
df = pd.get_dummies(data=df, columns=['AHD'], drop_first=True)
df['ChestPain'], ChestPain_codes = pd.factorize(df['ChestPain'])
df['Thal'], Thal_codes = pd.factorize(df['Thal'])
# Drop NA rows:
df.dropna(inplace=True)
df.reset_index(inplace=True)

# Split in train/test
X_train, X_test, y_train, y_test = train_test_split(
    df.drop(columns=['AHD_Yes']), df['AHD_Yes'],
    test_size=1/5, random_state=42)

# Tree controlls for large tree
tree_settings = {'criterion': 'entropy',
                 'min_samples_split': 2,
                 'min_samples_leaf': 1,
                 'min_impurity_decrease': 1e-10 ,
                 'random_state': 1}
```

Now we are ready to perform the bagging procedure.

[86]:
```
from sklearn import tree
from sklearn.utils import resample
import matplotlib.pyplot as plt

# Number of Bootstrap samples
B = 200
# Number of datapoints in each sample
n_train = X_train.shape[0]

# Empty DataFrames
cur_pred = pd.DataFrame(data=np.zeros((B, X_test.shape[0]))) * np.nan
pred = cur_pred.copy()

# Bootstrap loop
for b in range(B):
    # Resample train data
    samp_X, samp_y = resample(X_train, y_train, n_samples=n_train,
                              random_state=b)
    # Grow and fit tree
    cur_tree = tree.DecisionTreeClassifier().set_params(**tree_settings)
    cur_tree.fit(samp_X, samp_y)

    # Add current prediction of test data
    cur_pred.loc[b, :] = cur_tree.predict(X_test)
    # update prediction with Majority vote
    pred.loc[b, :] = cur_pred.mode().iloc[0, :]

# Test error
e_test = np.mean(abs(pred - y_test.reset_index(drop=True)), axis=1)

# display results
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(1, 1, 1)
plt.plot(np.arange(1, B+1), e_test, 'r-', label='Test error')
ax.set_xlabel('Bootstrap samples (b)')
ax.set_ylabel('test error')

plt.show()
```



8.4 Out-of-Bag Error

- There is a very straightforward way to estimate the test error of a bagged model, without the need to perform cross-validation or the validation set approach
- Recall that the key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations
- One can show that on average, each bagged tree makes use of around two-thirds of the observations
- The remaining one-third of the observations not used to fit a given bagged tree is referred to as the **out-of-bag (OOB)** observations
- We can predict the response for the ith observation using each of the trees in which that observation was OOB

8.4.1 Out-of-Bag Error Estimate

Let $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ be the training set and $\hat{f}_1^*, \ldots, \hat{f}_B^*$ be B bootstrapped classification methods (e.g. decision trees). 1. For $i = 1, \ldots, n$ find all bootstrapped models $\hat{f}_b^*$ that do not use the i-th observation for training. Use these models to make a prediction $\hat{y}_i^*$ by means of a majority vote 2. The out-of-bag (OOB) error estimate is the classification error

$$\text{Err}^* = \frac{1}{n} \sum_{i=1}^{n} I\left(y_i \neq \hat{y}_i^*\right).$$

- The resulting OOB error is a valid estimate of the test error for the bagged model, since the response for each observation is predicted using only the trees that were not fit using that observation

- It can be shown that with B sufficiently large, OOB error is virtually equivalent to leave-one-out cross-validation error
- Particularly convenient when performing bagging on large data sets for which cross-validation would be computationally onerous

8.4.2 Code Example Out-of-Bag Error

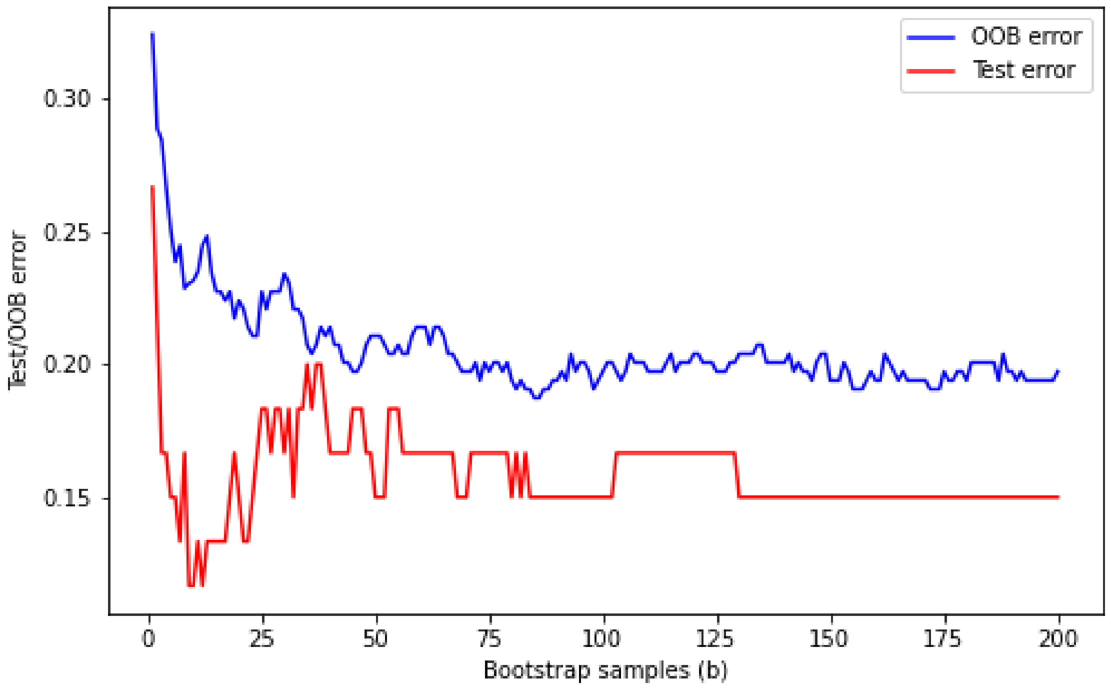
We compute the OOB error estimate for the **heart** data. Again, we perform the computations explicitly and we will discuss a more convenient computation by means of the **RandomForestClassifier()**-function later. We use the same number $B = 200$ of bootstrap samples as above, however there is no need of a training or test set, such that we redefine the initializations

[87]:
```
# Redifine x and y (no split)
X = df.drop(columns=['AHD_Yes'])
y = df['AHD_Yes']

# Number of datapoints in each sample
n = X.shape[0]

e_oob = []
pred = pd.Series(data=np.zeros(n)) * np.nan
cur_pred = pd.DataFrame(data=np.zeros((B, n))) * np.nan
```

The variable **cur_pred** contains B rows and n predictions. Each entry will only contain the out-of-bag samples and otherwise nan. The variable **pred** will contain a Series of length n which contains the *latest* majority vote. The OOB-error is then calculated using **pred** and the true class.

[88]:
```
# Bootstrap loop
for b in range(B):
    # Resample train data
    samp_X, samp_y = resample(X, y, n_samples=n,
                              random_state=b)
    # Store index OOB samples:
    mask = np.isin(X.index, samp_X.index, invert=True)
    i_oob = X.index[mask]
    # Grow and fit tree
    cur_tree = tree.DecisionTreeClassifier().set_params(**tree_settings)
    cur_tree.fit(samp_X, samp_y)

    # Add current prediction of oob data
    cur_pred.loc[b, i_oob] = cur_tree.predict(X.loc[i_oob, :])
    # update prediction with Majority vote
    pred.loc[i_oob] = cur_pred.loc[:, i_oob].mode().iloc[0, :]

    # Calculate OOB error over all oob samples
    cur_e_oob = np.mean(abs(pred - y))
    e_oob.append(cur_e_oob)
```

```python
# display results
fig = plt.figure(figsize=(8, 5))

ax = fig.add_subplot(1, 1, 1)
plt.plot(np.arange(1, B+1), e_oob, 'b-', label='OOB error')
plt.plot(np.arange(1, B+1), e_test, 'r-', label='Test error')

ax.set_xlabel('Bootstrap samples (b)')
ax.set_ylabel('Test/OOB error')
plt.legend()

plt.show()
```



As it can be seen, the OOB-error approximates the test error fairly well; without the need of a test set, though.

8.5 Random Forests

- Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees
- As in bagging, we build a number of decision trees on bootstrapped training samples.
- But when building these decision trees, each time a split in a tree is considered, a **random sample** of m **predictors is chosen** as split candidates from the full set of p predictors
- A split in the Decision Tree is allowed to use only one of those m predictors
- a fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$ that is,

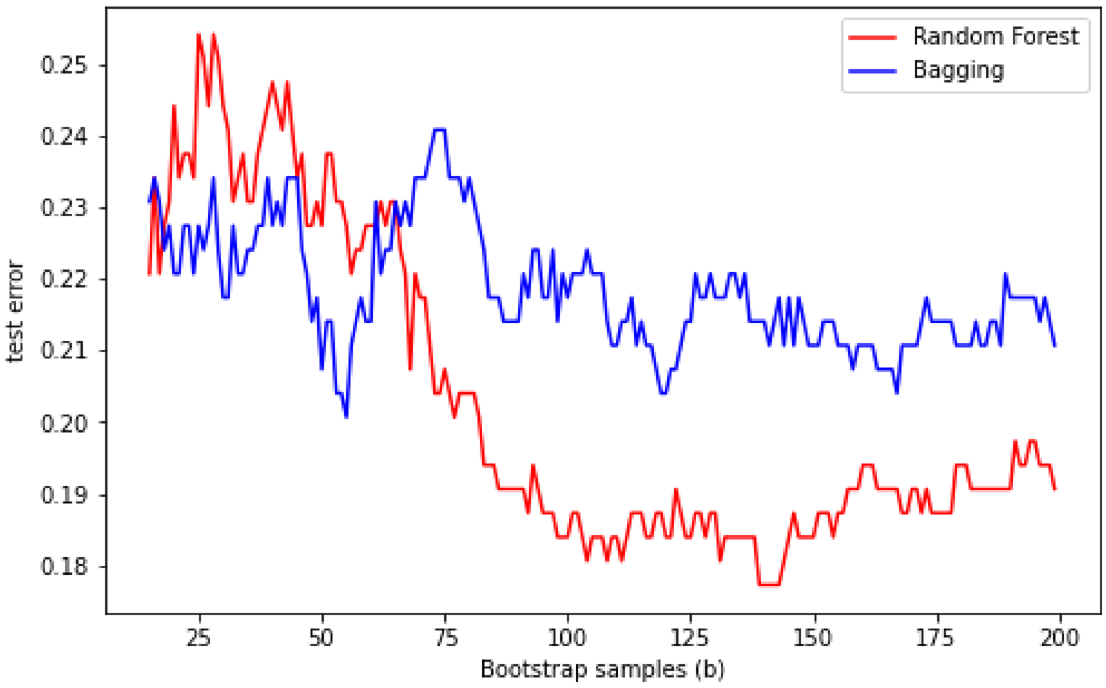
the number of predictions considered at each split is approximately equal to the square root of the total number of predictors
- Building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available predictors
- Clever rationale: Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors -> in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split
- Consequently, all of the bagged trees will look quite similar to each other : predictions from the bagged trees will be highly correlated
- **Random Forests overcome this problem by forcing each split to consider only a subset of the predictors**

8.5.1 Random Forests Variable Importance

- Bagging typically results in improved accuracy over prediction using a single tree. Unfortunately, it can be difficult to interpret the resulting model
- Thus, bagging improves prediction accuracy at the expense of interpretability
- One can obtain an overall summary of the importance of each predictor using the Gini index
- We can add up the total amount that the Gini index is decreased by splits over a given predictor, averaged over all B trees
- This averaged value is termed variable importance and tells us for each predictor how large its contribution to the model is

8.5.2 Code Example Random Forest

[89]:
```python
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv(path_data+'Heart.csv')

# Replace Categorical Variable with dummies
df = pd.get_dummies(data=df, columns=['AHD'], drop_first=True)
df['ChestPain'], ChestPain_codes = pd.factorize(df['ChestPain'])
df['Thal'], Thal_codes = pd.factorize(df['Thal'])
# Drop NA rows:
df.dropna(inplace=True)

X = df.drop(columns=['AHD_Yes'])
y = df['AHD_Yes']

n_features = X.shape[1]

# Tree controlls for RF
rfc_settings = {'oob_score': True,
                'max_features': np.sqrt(n_features)/n_features,
                'random_state': 1,
                'warm_start': True}
bag_settings = {'oob_score': True,
                'max_features': n_features,
                'random_state': 1,
                'warm_start': True}

# loop over B estimators
B = np.arange(15, 200, 1)

rfc = RandomForestClassifier().set_params(**rfc_settings)
bag = RandomForestClassifier().set_params(**bag_settings)

error_rfc = []
error_bag = []

for b in B:
    # Grow and fit tree
    rfc.set_params(n_estimators=b)
    bag.set_params(n_estimators=b)

    rfc.fit(X, y)
    bag.fit(X, y)

    # Record the OOB error
    error_rfc.append(1 - rfc.oob_score_)
    error_bag.append(1 - bag.oob_score_)

# display results
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(1, 1, 1)
plt.plot(B, error_rfc, 'r-', label='Random Forest')
plt.plot(B, error_bag, 'b-', label='Bagging')

ax.set_xlabel('Bootstrap samples (b)')
ax.set_ylabel('test error')
plt.legend()
plt.show()
```



The figure shows the OOB error estimates for a bagged tree (cyan) and a random forest with $m = \sqrt{p}$ (red) depending on the number of trees. It becomes clear that the random forest outperforms the bagged tree by approximately 3%.

8.5.3 Variable Importance Code Example

For the random forest model we can compute the variable importance as follows:

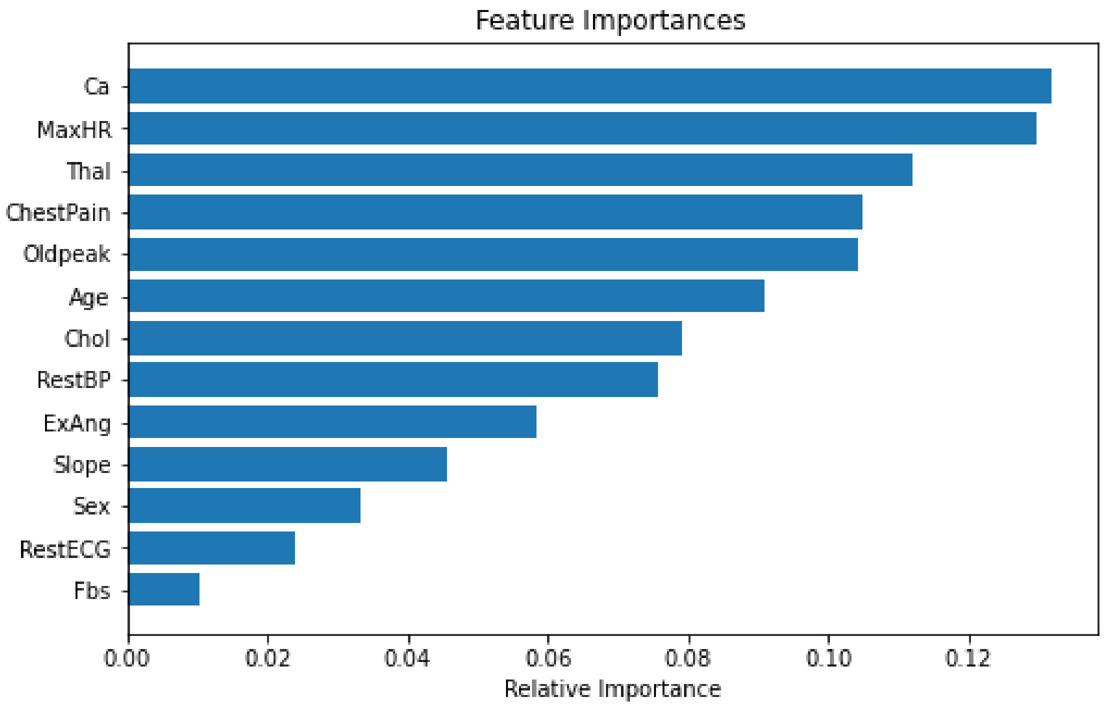
[90]:
```
# Feature Importances based onthe impurity decrease
importance = rfc.feature_importances_
features = X.columns.values

# Sort by importance
features = features[np.argsort(importance)]
importance = importance[np.argsort(importance)]

fi = pd.DataFrame({'feature': features,
                   'importance': importance})
print(fi)
```

```
      feature  importance
0         Fbs    0.010402
1     RestECG    0.023922
2         Sex    0.033263
3       Slope    0.045718
4       ExAng    0.058270
```

```
5       RestBP    0.075715
6         Chol    0.079140
7          Age    0.090973
8      Oldpeak    0.104287
9    ChestPain    0.104912
10        Thal    0.112000
11       MaxHR    0.129631
12          Ca    0.131768
```

As can be seen, the **Ca** variable is the most influential with an normalized average decrease in the Gini index of 13% followed by **MaxHR** and **Thal**.

[91]:
```python
# Visualize results
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(1, 1, 1)
plt.title('Feature Importances')
plt.barh(range(len(importance)), importance)
plt.yticks(range(len(importance)), features)
plt.xlabel('Relative Importance')
plt.show()
```



8.6 Code Example Decision Tree versus Random Forest

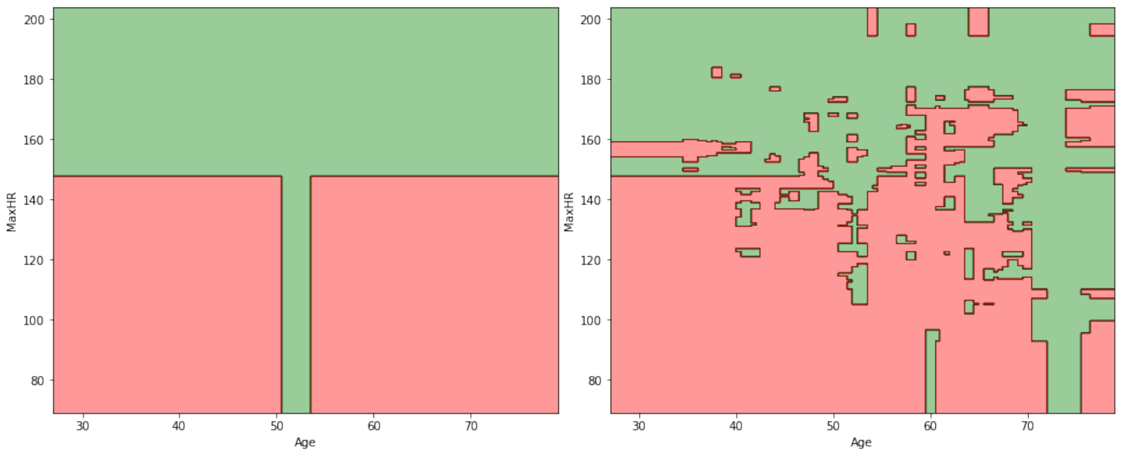
We again study the **heart** data but only consider the two predictors **age** and **maxHR**. We compute a random forest model and an ordinary tree and compare the partition of the two-dimensional predictor space according to where the models predit **AHD=yes** and **AHD=no**. We start by computing the models and defining a grid in the predictor space where we evaluate the resulting models. In fact, since we have only two predictors, we use a bagged model.

We define the variables, Tree-controls and create and fit the Decision-Tree and Random Forest model.

[92]:
```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv(path_data+'Heart.csv')

# Replace Categorical Variable with dummies
df = pd.get_dummies(data=df, columns=['AHD'], drop_first=True)
df['ChestPain'], ChestPain_codes = pd.factorize(df['ChestPain'])
df['Thal'], Thal_codes = pd.factorize(df['Thal'])
# Drop NA rows:
df.dropna(inplace=True)

X = df[['Age', 'MaxHR']]
y = df['AHD_Yes']

n_features = X.shape[1]

# Tree controlls for RF
rfc_settings = {'oob_score': True,
                'max_features': np.sqrt(n_features)/n_features,
                'random_state': 1,
                'n_estimators': 200}
tree_settings = {'criterion': 'entropy',
                 'min_samples_split': 10,
                 'min_samples_leaf': 5,
                 'min_impurity_decrease': 0.01,
                 'random_state': 1}

rfc = RandomForestClassifier().set_params(**rfc_settings)
tree = tree.DecisionTreeClassifier().set_params(**tree_settings)
rfc.fit(X, y)
tree.fit(X, y)
```

[92]:
```
DecisionTreeClassifier(criterion='entropy', min_impurity_decrease=0.01,
                       min_samples_leaf=5, min_samples_split=10,
                       random_state=1)
```

[93]:
```
# Create grid
margin, n = 2, 400
xx = np.linspace(min(X['Age']) - margin, max(X['Age']) + margin, n)
yy = np.linspace(min(X['MaxHR']) - margin, max(X['MaxHR']) + margin, n)
yy, xx = np.meshgrid(yy, xx)

# Predict
Ztree = tree.predict(np.c_[xx.ravel(), yy.ravel()])
Zrfc = rfc.predict(np.c_[xx.ravel(), yy.ravel()])
Ztree = Ztree.reshape(xx.shape)
Zrfc = Zrfc.reshape(xx.shape)

# Plots
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(14, 6))

# Plot surface
ax[0].contourf(xx, yy, Ztree,
               colors=['green', 'black', 'red', 'black'], alpha=0.4)
ax[1].contourf(xx, yy, Zrfc,
               colors=['green', 'black', 'red', 'black'], alpha=0.4)

ax[0].set_xlabel('Age'), ax[1].set_xlabel('Age')
ax[0].set_ylabel('MaxHR'), ax[1].set_ylabel('MaxHR')
plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)
plt.show()
```



9 Regularization Methods

- **Subset selection methods**: involve least squares to fit a linear model that contains a **subset** of the predictor variables
- Alternative: we fit a model containing all p predictors using a technique that **constrains or**

regularizes the coefficient estimates, or equivalently, that shrinks the coefficient estimates towards zero

- Shrinking the coefficient estimates can significantly reduce their variance
- Two best-known techniques for shrinking the regression coefficients towards zero are : ridge regression and the lasso equivalently that shrinks the coefficient estimates towards zero

9.1 Ridge Regression

- Least squares fitting procedure estimates $\beta_0, \beta_1, \ldots, \beta_p$ using the values that minimize

$$\text{RSS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2$$

- Ridge regression coefficient estimates are the $\hat{\beta}^R$ values that minimize

$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 = \text{RSS} + \lambda \sum_{j=1}^{p} \beta_j^2$$

  where $\lambda \geq 0$ is a tuning parameter or regularization parameter

- trades off two different criteria:
  - ridge regression (like least squares) seeks coefficients that fit the date well by making RSS small
  - the second term $\lambda \sum_j \beta_j^2$ is called **shrinkage penalty**
  - penalty is small when $\beta_1, \ldots, \beta_p$ are close to zero, and so it has effect of shrinking the estimates towards zero

9.1.1 Tuning Parameter

- tuning parameter serves to control the impact
  - $= 0$: penalty term has no effect. Equals the least squares estimate
  - $\to \infty$: regression coefficients approach 0

9.1.2 Disadvantages

- unlike best subset and stepwise (forward or backward) selection, ridge regression includes all p predictor variables in the final model
- penalty shrinks "only" **towards** 0 but never sets the coefficients exactly to 0
- model interpretability becomes difficult where p is large

9.2 Lasso

- overcomes disadvantages of ridge regression
- The lasso coefficients, $\hat{\beta}_\lambda^L$, minimize the quantity

$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| = \text{ RSS } + \lambda \sum_{j=1}^{p} |\beta_j|$$

  where $|\beta_j|$ corresponds to the L1-norm

- Only difference between ridge regression and the lasso is that the term β_j^2 in the ridge regression penalty has been replaced by $|\beta_j|$ in the lasso penalty
- $l1$ penalty of the lasso has the effect of forcing some of the coefficient estimates to be exactly equal to zero when the tuning parameter is sufficiently large
- Hence, much like best subset selection, the lasso performs **variable selection —+ increased interpretability**

9.2.1 Selecting tuning parameter

- **Cross-Validation** provides a simple way to tackle this problem
- Choose grid of values and compute cross validation error for each

9.3 Boosting

- Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification
- Boosting for decision trees: the trees are grown sequentially: each tree is grown using information from previously grown trees
- Each tree is fit on a modified version of the original data set

9.3.1 Boosting for regression trees

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \dots, B$, repeat:
    1. Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to the training data (X, r)
    2. Update $\hat{f}$ by adding in a shrunken version of the new tree:
    $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$
    3. Update the residuals,
    $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$
3. Output the boosted model,
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

- Given the current model, we fit a decision tree to the residuals from the model - instead of outcome Y, tree is fit to current residuals
- We then add this new decision tree into the fitted function in order to update the residuals
- Each of these trees can be rather small, with just a few terminal nodes - determined by parameter d in the algorithm. By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well
- Shrinkage parameter slows the process down even further, allowing more and different shaped trees to attack the residuals

9.4 Feature importance

- *Examples of interpretable models*: Best subset selection, Decision Trees, and the Lasso
- *Examples of model-specific interpretation methods*: Mean Decrease of Gini index in random forests and boosting

- *Model-agnostic feature importance methods*: If we separate the feature importance measures from the machine learning model
- *Model flexibility*: The interpretation method can work with any machine learning model, such as random forests and deep neural networks.
- *Explanation flexibility*: You are not limited to a certain form of explanation. In some cases it might be useful to have a linear formula, in other cases a graphic with feature importances.
- *Representation flexibility*: The explanation system should be able to use a different feature representation as the model being explained.

9.4.1 Permutation feature importance

- a feature is important if shuffeling its values increases the model error
  - swapping the value of two coefficients for example
- it is unimportant if shuffeling leaves the model error unchanged

Permutation feature importance

Input: Trained model $\hat{f}$, measurements x_i in rows of data matrix X, response variable y, error measure $L(y, \hat{f}(X)\ )$ (e.g. mean squared error) 1. Estimate the original model error $L(y, \hat{f}(X))$ 2. For each feature $j \in \{1, 2, \ldots, p\}$ do: 1. Generate data matrix X_{perm} by permuting feature j in the data matrix X. This breaks the association between feature j and the true outcome y. 2. Estimate error $e_{\text{perm}} = L\left(y, \hat{f}\left(X_{\text{perm}}\right)\right)$ based on the predictions of the permuted data. 3. Calculate permutation feature importance as quotient $Fl_j = e_{\text{perm}}/e_{\text{orig}}$ or as difference $Fl_j = e_{\text{perm}} - e_{\text{orig}}$ 3. Sort features j by descending Fl_j

9.4.2 Computing the feature importance on training or test data?

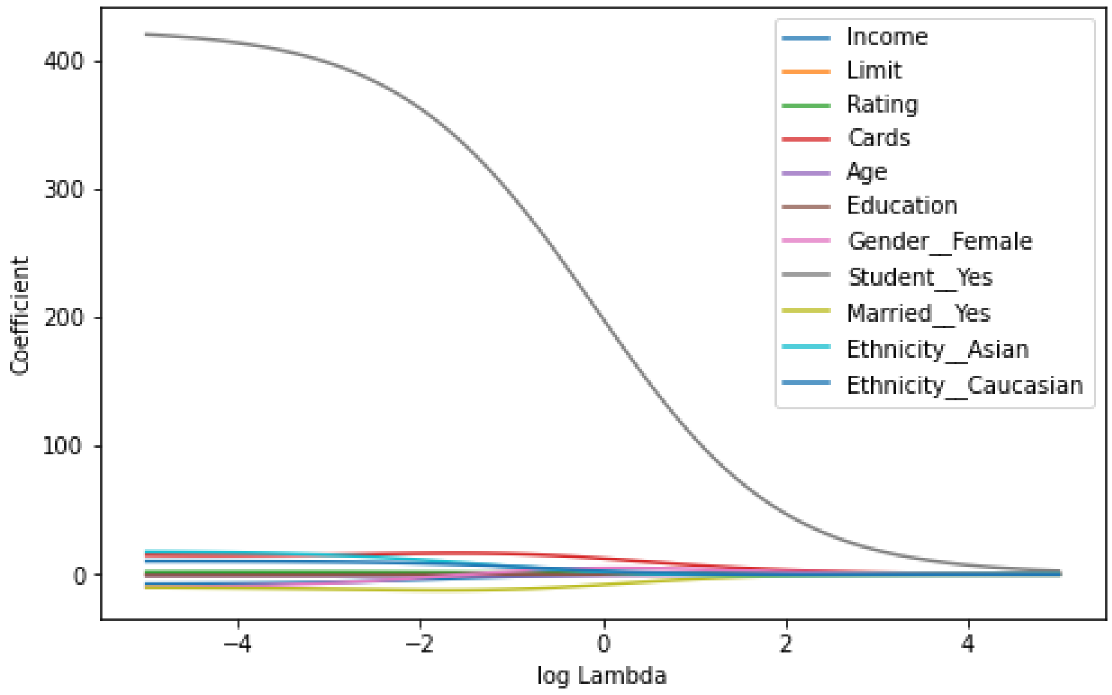
- Case for test data:
  - General principle in machine learning: If we measure the model error (or performance) on the same data on which the model was trained, the measurement is usually too optimistic
  - The feature importance based on training data makes us mistakenly believe that features are important for the predictions, when in reality the model was just overfitting and the features were not important at all
- Case for training data
  - In practice, we want to use all our data to train our model to get the best possible model in the end.
  - We are interested in how much the trained model's predictions are influenced by a feature.

9.5 Code examples Regularization

9.5.1 Ridge

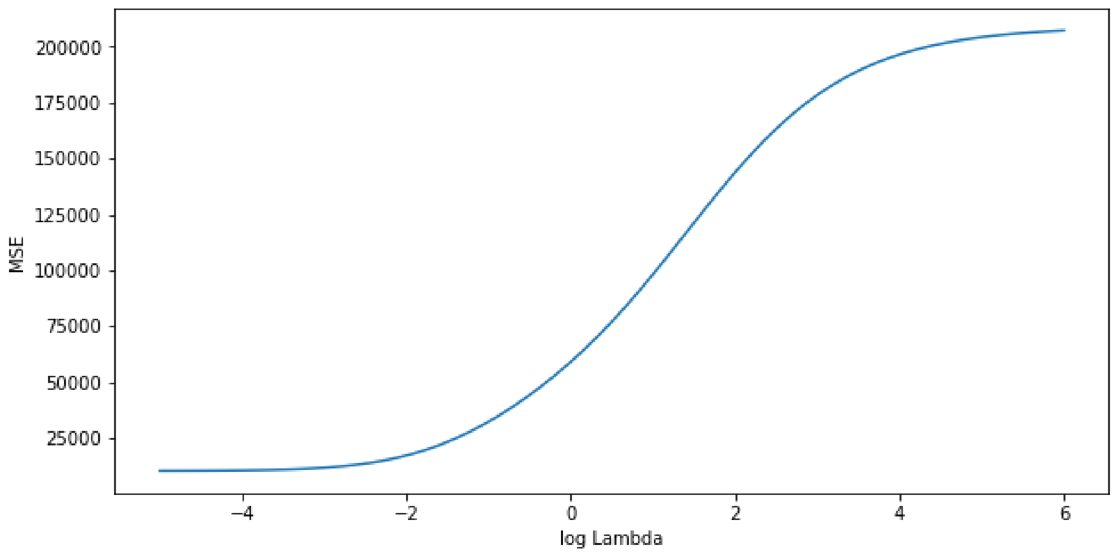
[94]:
```
import pandas as pd
import numpy as np
from sklearn.linear_model import Ridge
import warnings
warnings.filterwarnings("ignore")
```

```
# Load data
df = pd.read_csv(path_data+'Credit.csv', index_col="Unnamed: 0")

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

# Define target and predictors
x = df.drop(columns='Balance')
y = df['Balance']

# Model for different lambda
n = 100
lambda_ = np.exp(np.linspace(-5, 5, n))

params = pd.DataFrame(columns=x.columns)
for i in range(n):
    reg = Ridge(alpha=lambda_[i], normalize=True)
    reg = reg.fit(x, y)
    params.loc[np.log(lambda_[i]), :] = reg.coef_


import matplotlib.pyplot as plt
# Plot
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(1, 1, 1)
params.plot(ax=ax)
plt.xlabel("log Lambda")
plt.ylabel("Coefficient")
plt.legend()
plt.show()
```



RidgeCV

[95]:
```
import pandas as pd
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split
from sklearn.linear_model import RidgeCV
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv(path_data+'Credit.csv', index_col="Unnamed: 0")

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

# Define target and predictors
x = df.drop(columns='Balance')
y = df['Balance']

# Split in test and train set
np.random.seed(10)
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                                                     test_size=0.5)

n=100
lambda_ = np.exp(np.linspace(-5, 6, n))

# Fit model:
reg = RidgeCV(alphas=lambda_, store_cv_values=True, normalize=True)
reg = reg.fit(x_train, y_train)

# Plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(np.log(lambda_), np.mean(reg.cv_values_, axis=0))
plt.xlabel("log Lambda")
plt.ylabel("MSE")
plt.show()
```



[96]:
```
print("Best Lambda:", np.round(reg.alpha_, 3))
```

```
Best Lambda: 0.007
```

[97]:
```
# Predict and calculate MSE on test-set
y_pred = reg.predict(x_test)
MSE = np.mean((y_pred - y_test)**2)
print("MSE:", np.round(MSE, 1))
```

```
MSE: 10894.1
```

[98]:
```python
# Coeficient and coresponding predictors
coef = np.round(reg.coef_, 3)
x_cols = x.columns.values

print(pd.DataFrame(data={'Feature': x_cols,
                         'Coefficient':coef}))
```

```
                Feature  Coefficient
0                Income       -7.406
1                 Limit        0.139
2                Rating        1.803
3                 Cards       18.373
4                   Age       -0.382
5             Education        0.984
6        Gender__Female       13.514
7          Student__Yes      451.436
8          Married__Yes      -10.541
9      Ethnicity__Asian       20.025
10 Ethnicity__Caucasian       28.396
```

9.5.2 Lasso

[99]:
```python
import pandas as pd
import numpy as np
from sklearn.linear_model import Lasso
import warnings
warnings.filterwarnings("ignore")

# Load data
df = pd.read_csv(path_data+'Credit.csv', index_col="Unnamed: 0")

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

# Define target and predictors
x = df.drop(columns='Balance')
y = df['Balance']

# Fit model:
lambda_ = 5
reg = Lasso(alpha=lambda_, normalize=True)
reg = reg.fit(x, y)

# Coefficient and corresponding predictors
coef = np.round(reg.coef_, 3)
```

```
x_cols = x.columns.values
```

We expect the coefficient estimates to be much smaller, in terms of ℓ_2 norm, when a large value of λ is used, as compared to when a small value of λ is used. These are the coefficients when $\lambda = 5$, along with their ℓ_2 norm:

```
[100]: print(pd.DataFrame(data={'Feature': x_cols,
                              'Coefficient':coef}),
             '\n\nl2-norm:', np.sqrt(np.sum(coef**2)))
```

```
                 Feature  Coefficient
0                 Income       -0.000
1                  Limit        0.010
2                 Rating        1.769
3                  Cards        0.000
4                    Age       -0.000
5              Education        0.000
6         Gender__Female        0.000
7           Student__Yes       65.438
8           Married__Yes       -0.000
9       Ethnicity__Asian        0.000
10  Ethnicity__Caucasian       -0.000

l2-norm: 65.46190728202167
```

In contrast, here are the coefficients when $\lambda = 0.5$, along with their ℓ_2 norm. Note the much higher ℓ_2 norm of the coefficients associated with this lower value of λ.

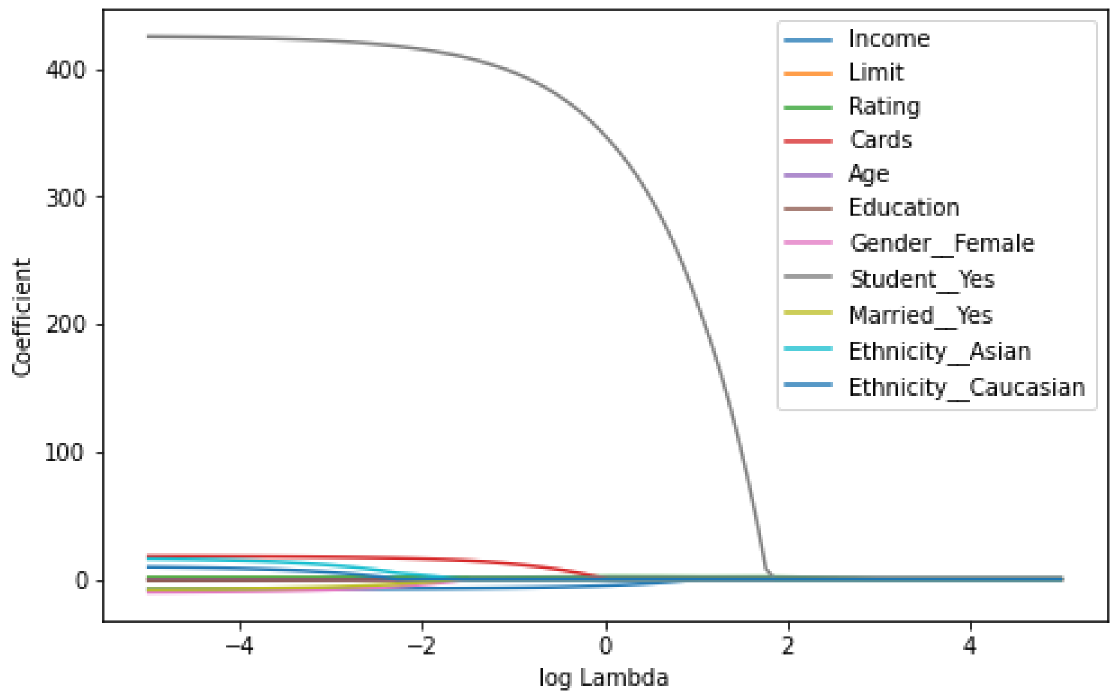
```
[101]: # Fit model:
       lambda_ = 0.5
       reg = Lasso(alpha=lambda_, normalize=True)
       reg = reg.fit(x, y)

       # Coefficient and corresponding predictors
       coef = np.round(reg.coef_, 3)
       x_cols = x.columns.values

       print(pd.DataFrame(data={'Feature': x_cols,
                              'Coefficient':coef}),
             '\n\nl2-norm:', np.sqrt(np.sum(coef**2)))
```

```
                 Feature  Coefficient
0                 Income       -6.491
1                  Limit        0.145
2                 Rating        1.508
3                  Cards        9.417
4                    Age       -0.236
5              Education       -0.000
```

```
6          Gender__Female       -0.000
7            Student__Yes      387.093
8            Married__Yes       -0.000
9        Ethnicity__Asian        0.000
10   Ethnicity__Caucasian        0.000

l2-norm: 387.26496666236153
```

As we can observe, several coefficients have become zero. The larger the value of λ the more coefficients become zero. Variable selection thus is generically implemented in the lasso.

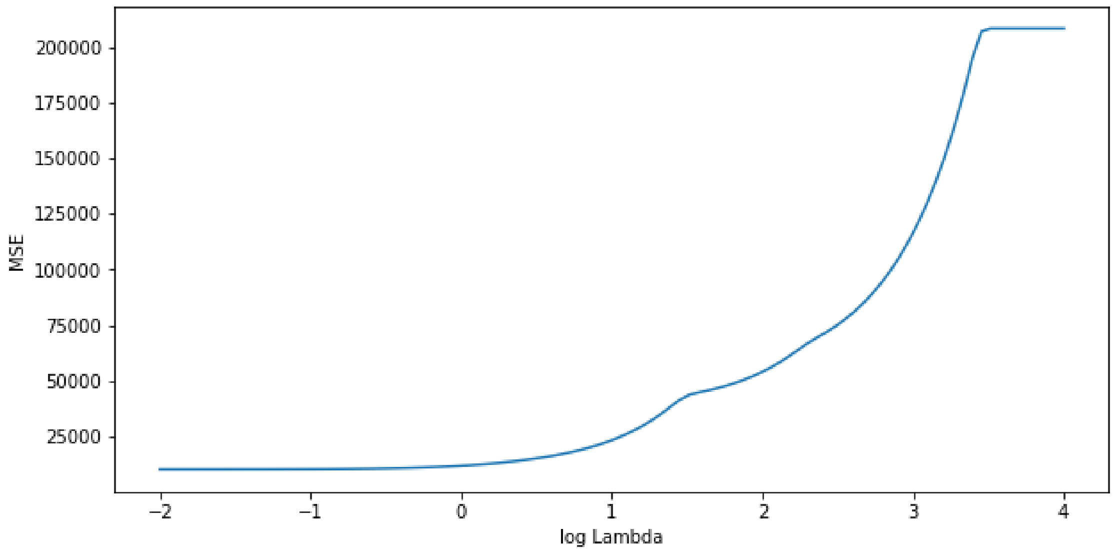
[102]:
```python
import matplotlib.pyplot as plt

# Model for different lambda
n = 100
lambda_ = np.exp(np.linspace(-5, 5, n))

params = pd.DataFrame(columns=x.columns)
for i in range(n):
    reg = Lasso(alpha=lambda_[i], normalize=True)
    reg = reg.fit(x, y)
    params.loc[np.log(lambda_[i]), :] = reg.coef_

# Plot
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(1, 1, 1)
params.plot(ax=ax)
plt.xlabel("log Lambda")
plt.ylabel("Coefficient")
plt.legend()
plt.show()
```



Lasso CV

[103]:
```python
import pandas as pd
import numpy as np
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.linear_model import LassoCV

# Load data
df = pd.read_csv(path_data+'Credit.csv', index_col="Unnamed: 0")

# Convert Categorical variables
df = pd.get_dummies(data=df, drop_first=True,
                    prefix=('Gender_', 'Student_',
                            'Married_', 'Ethnicity_'))

# Define target and predictors
x = df.drop(columns='Balance')
y = df['Balance']

# Split in test and train set
np.random.seed(10)
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                                    test_size=0.5)
```

```
lambda_ = np.exp(np.linspace(-2, 4, n))

# Fit model:
reg = LassoCV(alphas=lambda_, normalize=True)
reg = reg.fit(x_train, y_train)

# Plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(np.log(reg.alphas_), np.mean(reg.mse_path_, axis=1))
plt.xlabel("log Lambda")
plt.ylabel("MSE")
plt.show()
```



We can see from the coefficient plot that depending on the choice of tuning parameter, some of the coefficients will be exactly equal to zero. We now compute the associated test error.

[104]:
```
# Predict and calculate MSE on test-set
y_pred = reg.predict(x_test)
MSE = np.mean((y_pred - y_test)**2)
print("Best Lambda:", np.round(reg.alpha_, 3), "\nMSE:", np.round(MSE, 1))
```

```
Best Lambda: 0.135
MSE: 10538.4
```

This is substantially lower than the test set MSE of the null model, the test MSE of ridge regression with λ chosen by cross-validation, and very similar to least squares. Finally, we will have a look at the corresponding coefficients.

[105]:
```
# Coeficient and coresponding predictors
coef = np.round(reg.coef_,  3)
x_cols = x.columns.values

print(pd.DataFrame(data={'Feature': x_cols,
                         'Coefficient':coef}))
```

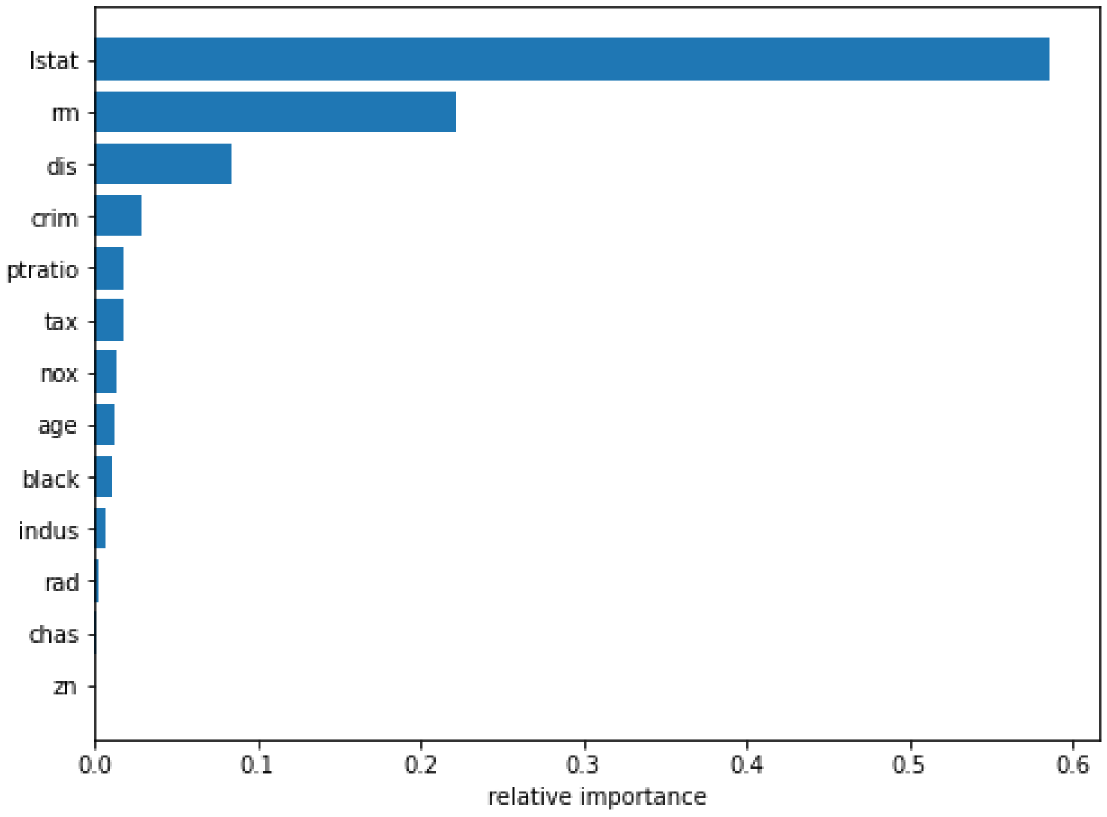
```
                 Feature  Coefficient
0                 Income       -7.410
1                  Limit        0.166
2                 Rating        1.392
3                  Cards       18.422
4                    Age       -0.277
5              Education        0.111
6         Gender__Female        8.978
7           Student__Yes      448.564
8           Married__Yes       -4.421
9       Ethnicity__Asian        7.698
10  Ethnicity__Caucasian       17.483
```

9.5.3 Gradient boosting

Here we use **GradientBoostingRegressor()** from **sklearn.ensemble**, to fit a boosted regression trees model to the Boston data set. The argument **n__estimators=5000** indicates that we want 5000 trees, the option **max__depth=4** limits the depth of each tree, and **learning__rate=0.001** shrinks the contribution of each tree.

[106]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor

# Load data
df = pd.read_csv(path_data+'Boston.csv', index_col=0)

# Define target and predictors
x = df.drop(columns='medv')
y = df['medv']

# Split in test and train set
np.random.seed(1)
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                                    test_size=0.5)

# Fit model:
reg = GradientBoostingRegressor(n_estimators=5000, max_depth=4,
                                learning_rate=0.001)
reg = reg.fit(x_train, y_train)
```

[107]:
```python
import matplotlib.pyplot as plt

# Feature Importances based on the impurity decrease
importance = reg.feature_importances_
features = x_train.columns.values

# Sort by importance
features = features[np.argsort(importance)]
importance = importance[np.argsort(importance)]

# plot
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.barh(features, importance)
ax.set_xlabel("relative importance")
plt.show()
```



lstat and **rm** by far the most important variables

We now use the boosted model to predict **medv** on the test set:

```
[108]: from sklearn.metrics import mean_squared_error

       # Predict
       pred = reg.predict(x_test)

       # MSE
       MSE = mean_squared_error(y_test, pred)
       print("MSE:", np.round(MSE, 3))
```

```
MSE: 10.675
```

The test MSE obtained is 10.361; similar to the test MSE for random forests and superior to that for bagging. If we want to, we can perform boosting with a different value of the shrinkage parameter λ. The default value is 0.1, but this is easily modified. Here we take $\lambda = 0.2$.

```
[109]: # Fit model with higher learning rate:
       reg = GradientBoostingRegressor(n_estimators=5000, max_depth=4, learning_rate=0.
        ↪2)
       reg = reg.fit(x_train, y_train)

       # Predict
       pred = reg.predict(x_test)

       # MSE
       MSE = mean_squared_error(y_test, pred)
       print("MSE:", np.round(MSE, 3))
```

```
MSE: 10.228
```

In this case, using $\lambda = 0.2$ leads to a slightly lower test MSE than $\lambda = 0.001$.

9.5.4 Feature importance

```
[110]: import matplotlib.pyplot as plt
       import numpy as np

       from sklearn.datasets import fetch_openml
       from sklearn.ensemble import RandomForestClassifier
       from sklearn.impute import SimpleImputer
       from sklearn.inspection import permutation_importance
       from sklearn.compose import ColumnTransformer
       from sklearn.model_selection import train_test_split
       from sklearn.pipeline import Pipeline
       from sklearn.preprocessing import OneHotEncoder
```

Let's use pandas to load a copy of the titanic dataset. The following shows how to apply separate preprocessing on numerical and categorical features.

We further include two random variables that are not correlated in any way with the target variable (survived):

- `random_num` is a high cardinality numerical variable (as many unique values as records).
- `random_cat` is a low cardinality categorical variable (3 possible values).

[111]:
```
X, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)
rng = np.random.RandomState(seed=42)
X["random_cat"] = rng.randint(3, size=X.shape[0])
X["random_num"] = rng.randn(X.shape[0])

categorical_columns = ["pclass", "sex", "embarked", "random_cat"]
numerical_columns = ["age", "sibsp", "parch", "fare", "random_num"]

X = X[categorical_columns + numerical_columns]

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,␣
 ↪random_state=42)

categorical_encoder = OneHotEncoder(handle_unknown="ignore")
numerical_pipe = Pipeline([("imputer", SimpleImputer(strategy="mean"))])

preprocessing = ColumnTransformer(
    [
        ("cat", categorical_encoder, categorical_columns),
        ("num", numerical_pipe, numerical_columns),
    ]
)

rf = Pipeline(
    [
        ("preprocess", preprocessing),
        ("classifier", RandomForestClassifier(random_state=42)),
    ]
)
rf.fit(X_train, y_train)
```

[111]:
```
Pipeline(steps=[('preprocess',
                 ColumnTransformer(transformers=[('cat',
OneHotEncoder(handle_unknown='ignore'),
                                                  ['pclass', 'sex', 'embarked',
                                                   'random_cat']),
                                                 ('num',
                                                  Pipeline(steps=[('imputer',
SimpleImputer())]),
                                                  ['age', 'sibsp', 'parch',
                                                   'fare', 'random_num'])])),
                ('classifier', RandomForestClassifier(random_state=42))])
```

Accuracy of the Model

Prior to inspecting the feature importances, it is important to check that the model predictive performance is high enough. Indeed there would be little interest of inspecting the important features of a non-predictive model.

Here one can observe that the train accuracy is very high (the forest model has enough capacity to completely memorize the training set) but it can still generalize well enough to the test set thanks to the built-in bagging of random forests.

It might be possible to trade some accuracy on the training set for a slightly better accuracy on the test set by limiting the capacity of the trees (for instance by setting `min_samples_leaf=5` or `min_samples_leaf=10`) so as to limit overfitting while not introducing too much underfitting.

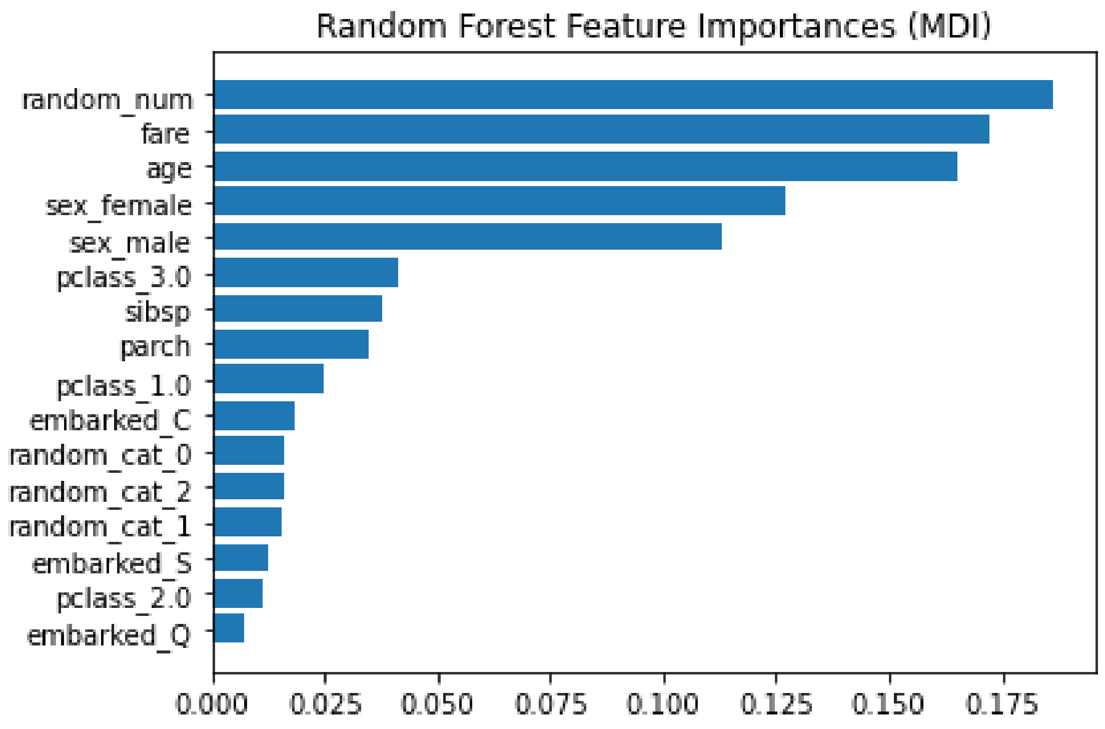
However let's keep our high capacity random forest model for now so as to illustrate some pitfalls with feature importance on variables with many unique values.

[112]:
```
print("RF train accuracy: %0.3f" % rf.score(X_train, y_train))
print("RF test accuracy: %0.3f" % rf.score(X_test, y_test))
```

```
RF train accuracy: 1.000
RF test accuracy: 0.817
```

Tree's Feature Importance from Mean Decrease in Impurity (MDI)

[113]:
```
ohe = rf.named_steps["preprocess"].named_transformers_["cat"]
feature_names = ohe.get_feature_names(categorical_columns)
feature_names = np.r_[feature_names, numerical_columns]

tree_feature_importances = rf.named_steps["classifier"].feature_importances_
sorted_idx = tree_feature_importances.argsort()

y_ticks = np.arange(0, len(feature_names))
fig, ax = plt.subplots()
ax.barh(y_ticks, tree_feature_importances[sorted_idx])
ax.set_yticks(y_ticks)
ax.set_yticklabels(feature_names[sorted_idx])
ax.set_title("Random Forest Feature Importances (MDI)")
fig.tight_layout()
plt.savefig('rf_importance_mdi.png')
plt.show()
```

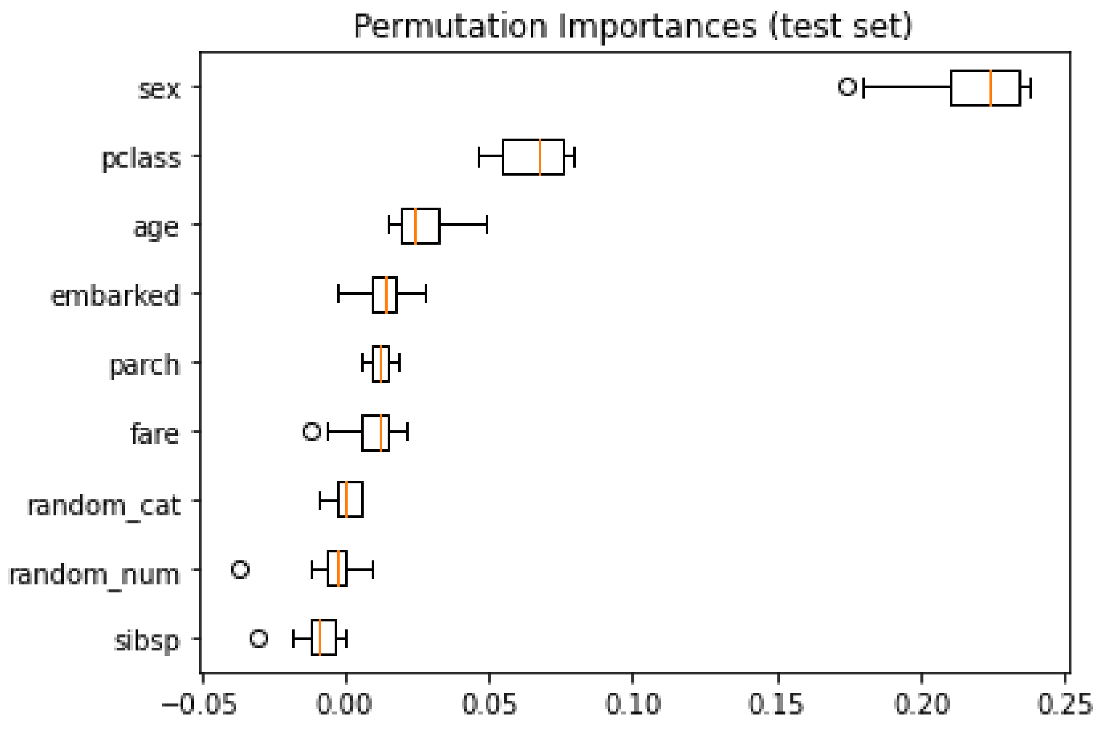


The impurity-based feature importance ranks the numerical features to be the most important features. As a result, the non-predictive `random_num` variable is ranked the most important!

This problem stems from two limitations of impurity-based feature importances:

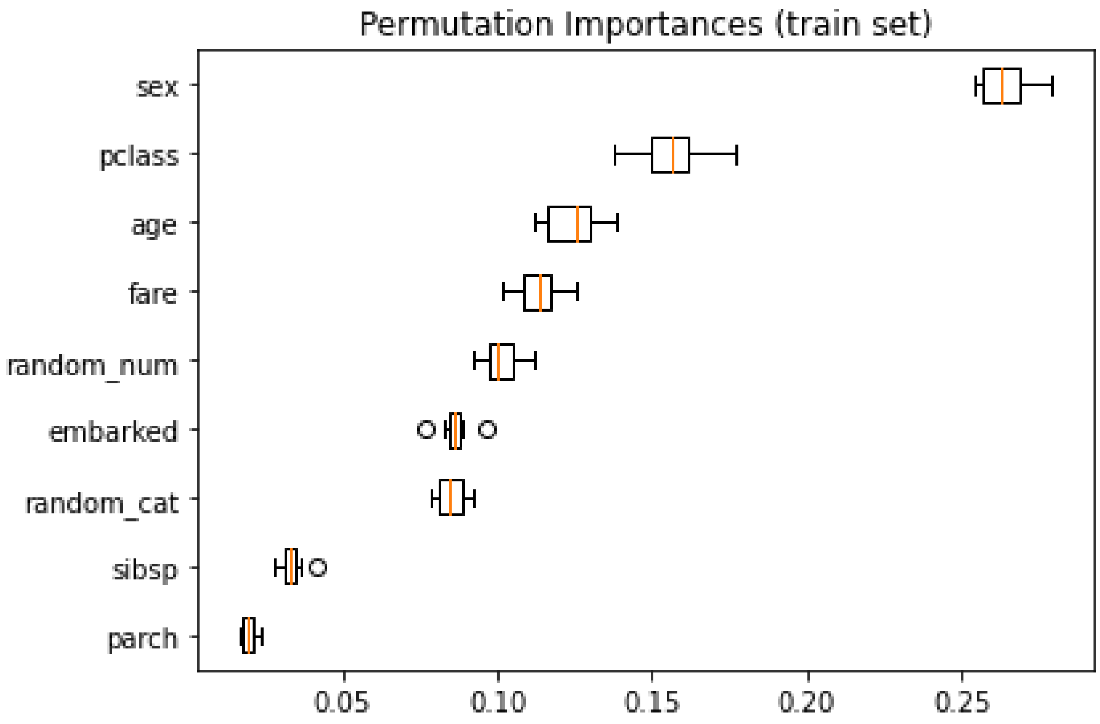
- impurity-based importances are biased towards high cardinality features;
- impurity-based importances are computed on training set statistics and therefore do not reflect the ability of feature to be useful to make predictions that generalize to the test set (when the model has enough capacity).

[114]:
```
result = permutation_importance(rf, X_test, y_test, n_repeats=10,␣
 ↪random_state=42)
sorted_idx = result.importances_mean.argsort()

fig, ax = plt.subplots()
ax.boxplot(
    result.importances[sorted_idx].T, vert=False, labels=X_test.
 ↪columns[sorted_idx]
)
ax.set_title("Permutation Importances (test set)")
fig.tight_layout()
plt.savefig('RT_Example_5_1_d.png')
plt.show()
```



It is also possible to compute the permutation importances on the training set. This reveals that `random_num` gets a significantly higher importance ranking than when computed on the test set. The difference between those two plots is a confirmation that the RF model has enough capacity to use that random numerical feature to overfit. You can further confirm this by re-running this example with constrained RF with `min_samples_leaf=10`.

```
[115]: result = permutation_importance(
           rf, X_train, y_train, n_repeats=10, random_state=42
       )
       sorted_idx = result.importances_mean.argsort()

       fig, ax = plt.subplots()
       ax.boxplot(
           result.importances[sorted_idx].T, vert=False, labels=X_train.
         ↪columns[sorted_idx]
       )
       ax.set_title("Permutation Importances (train set)")
       fig.tight_layout()
       plt.show()
```



10 Time Series Analysis

There are several goals that one wants to achieve with time series analysis . * **Descriptive Analysis**:

```
By means of summary statistics and visualizations, the basic properties of a time series are un
```

- **Modeling and Interpretation**:

  By modeling the underlying process that governs the observed time series, a deeper understanding can be gained. In particular, tests and confidence intervalls/bands can be constructed from the model. The sequential dependency of the time series is also often quantified. There are several goals that one wants to achieve with time series

- **Decomposition:**

  - Seasonality, in particular a periodic pattern in the data
  - Trend, gradually changing average of the series which is directly correlated with the time axis

- **Prediction**:

  By means of the model, future values of the time series can be predicted. Prediction for time series is often alternatively termed **forecasting**

- **Regression**:

One often tries to explain a time series (response) by several other time series (predictors). This idea is wide-spread in industry, where the goal is to replace in a multi-sensor setup a particular (expensive or hard to install) sensor by a model that predicts its values from the other sensor values. This is called *virtual sensoring* or *soft sensoring*

10.1 Basic transformation and Visualization of Time Series

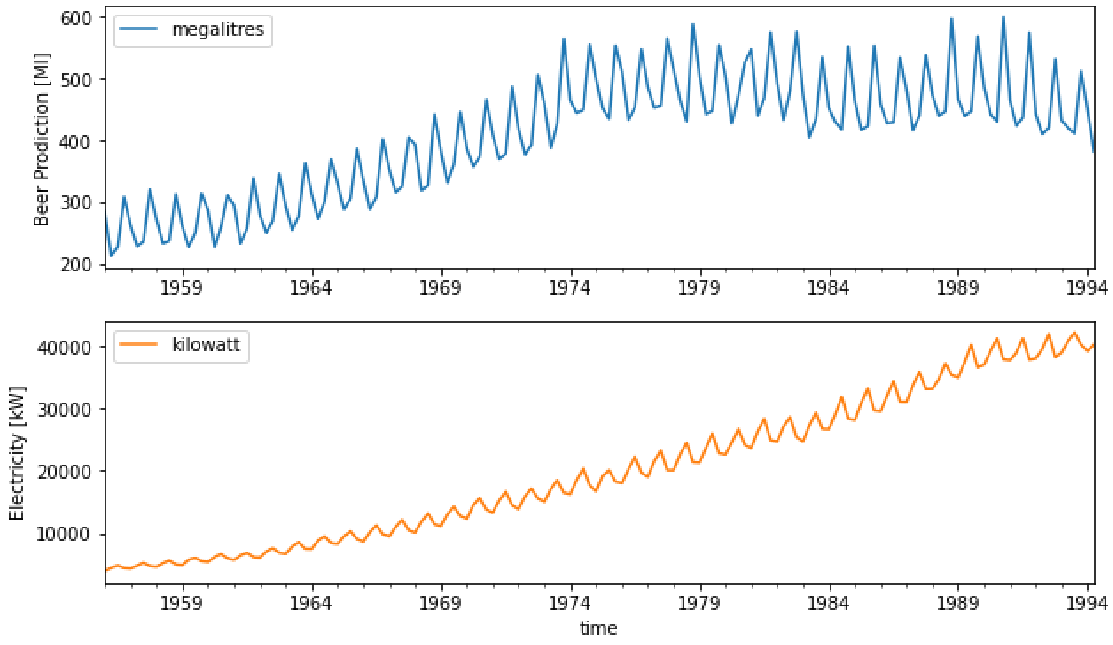
- Analysis of time series begins with the description, transformation and visualization of data → No modeling
- Does not give rise to proper predictions, confidence intervalls etc.
- Important insights and a profound understanding of the data can be achieved by these techniques
- Here:
  - Most important data transformations in the context of time series
  - Toolbox of helpful visualization techniques for exploring time series
  - Decomposition of time series into seasonal, trend and irregular components

10.1.1 Code Example Time Series

[116]:
```python
import numpy as np
import pandas as pd

# Load data
AusBeer = pd.read_csv(path_data+'AustralianBeer.csv', sep = ";", header = 0)
AusEl = pd.read_csv(path_data+'AustralianElectricity.csv', sep = ";")

# Create pandas DateTimeIndex
dtindexB = pd.DatetimeIndex(data=pd.to_datetime(AusBeer["Quarter"]),
                            freq='infer')
dtindexE = pd.DatetimeIndex(data=pd.to_datetime(AusEl["Quarter"]),
                            freq='infer')

# Set as Index
AusBeer.set_index(dtindexB, inplace=True)
AusEl.set_index(dtindexE, inplace=True)
AusBeer.drop("Quarter", axis=1, inplace=True)
AusEl.drop("Quarter", axis=1, inplace=True)

# Create new DataFrame combining both
Aus = pd.merge(AusBeer, AusEl, on="Quarter")

print(Aus.head(3))

import matplotlib.pyplot as plt

# Plot
fig, ax = plt.subplots(figsize=(10, 6), nrows=2)
# All data
```

```
Aus.plot(ax=ax[:],  subplots=True)

ax[0].set_xlabel("")
ax[1].set_xlabel("time")
ax[0].set_ylabel("Beer Prodiction [Ml]")
ax[1].set_ylabel("Electricity [kW]")

plt.show()
```

```
            megalitres  kilowatt
Quarter
1956-01-01       284.4      3923
1956-04-01       212.8      4436
1956-07-01       226.9      4806
```



10.1.2 Transformation of Time Series Data

- Usually desired, even necessary to transform time series before application of a model
  - In partiular, many methods assume a:
    - **Gaussian** or at least a **symmetric distribution** of the data
    - **Linear trend** relationship between time and data
    - **Constant variance** across time
- Example: for highly skewed or heteroskedastic (variance of the residual term, or error term, in a regression model varies widely) data, it is often better not to use the original series

$$\{X_1, X_2, ...\}$$

- but a transformed series
$$\{g(x_1), g(x_2), ...\}$$

Box-Cox-Transformation

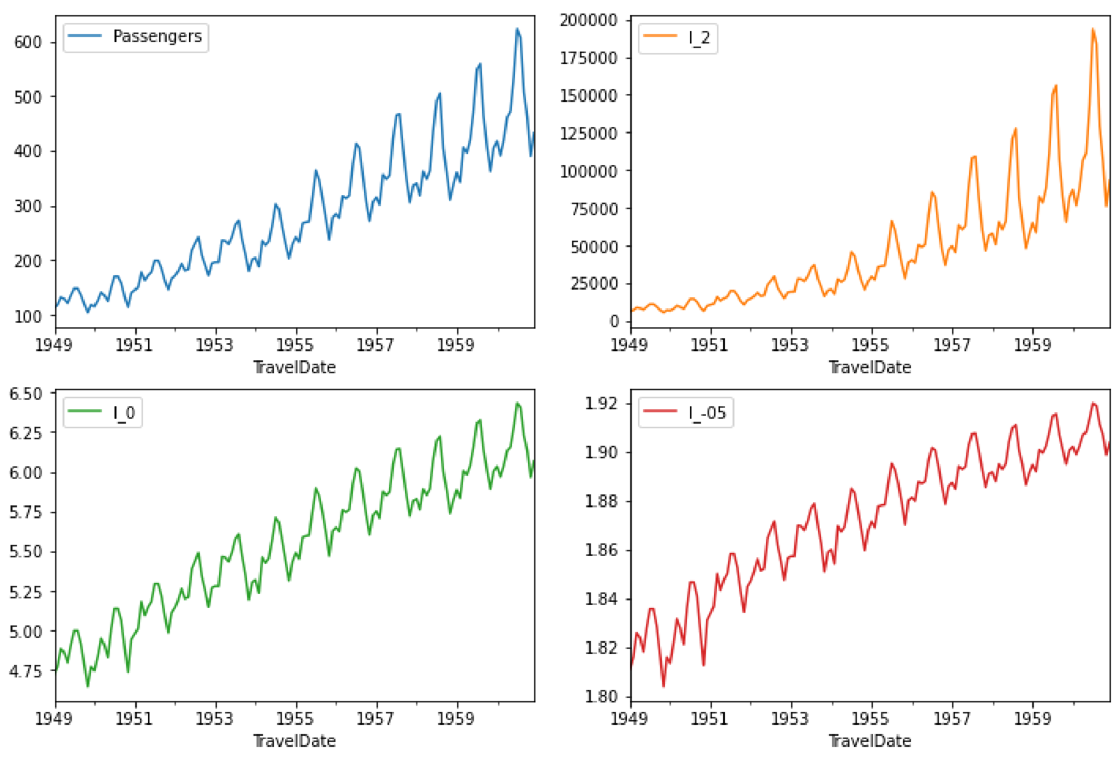
- correcting skewness and variance
- For a time series $\{x_1, x_2, ...\}$ with positive values the Box-Cox transformations are defined as

$$g(x) = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(x) & \text{if } \lambda = 0 \end{cases}$$

- choose the parameter such that the desired properties hold

Code Example Box-Cox Transformation

[118]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load data
AirP = pd.read_csv(path_data+'AirPassengers.csv', parse_dates=True)

# Create pandas DateTimeIndex
dtindex = pd.DatetimeIndex(data=pd.to_datetime(AirP["TravelDate"]),
                           freq='infer')
# Set as Index
AirP.set_index(dtindex, inplace=True)
AirP.drop("TravelDate", axis=1, inplace=True)

# Boxcox Definition:
def boxcox(x, lambd):
    if (lambd == 0):
        return np.log(x)
    else:
        return (x ** lambd - 1) / lambd

# Apply transform
AirP["l_2"] = boxcox(AirP["Passengers"], 2)
AirP["l_0"] = boxcox(AirP["Passengers"], 0)
AirP["l_-05"] = boxcox(AirP["Passengers"], -.5)

# Plots
fig, ax = plt.subplots(figsize=(12, 8), nrows=2, ncols=2)

AirP.plot(ax=ax[:],  subplots=True)
plt.show()
```



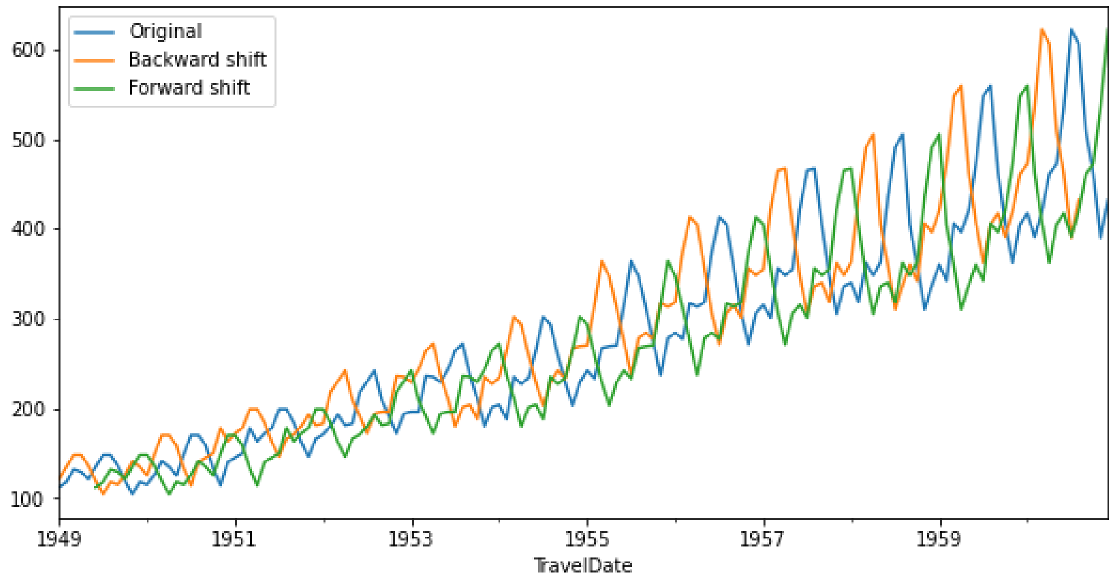
Time-Shift Transformation

- Box-Cox transforms the *values* of TS
- Time-Shift transforms the *time-axis*, which is sometimes necessary
- simplest is: *shifting*
- Let ${ x_1,x_2,… } $ be a time series:
  1. Time-shift by a *lag* of $k \in \mathbb{Z}$ is defined by
  $$g(x_i) = x_{i-k}$$
  2. For the particular case where $k = 1$ the time-shift is called *backshift*
  $$B(x_i) = x_{i-1}$$
- applying a time-shift to a time series amounts to go **back** k steps ($k > 0$) or go **ahead** ($k < 0$)
  - **Attention**: in python code the .shift() operation on a dataframe is reversed:
    * $k > 0 =$ shift ahead
    * $k < 0 =$ shift backwards

Code Example Time Shift

[119]:
```
import numpy as np
import pandas as pd
```

```python
import matplotlib.pyplot as plt

# Load data
AirP = pd.read_csv(path_data+'AirPassengers.csv', parse_dates=True)

# Create pandas DateTimeIndex
dtindex = pd.DatetimeIndex(data=pd.to_datetime(AirP["TravelDate"]),
                           freq='infer')
# Set as Index
AirP.set_index(dtindex, inplace=True)
AirP.drop("TravelDate", axis=1, inplace=True)

# Perform shift
AirP["s_4"] = AirP["Passengers"].shift(-4)
AirP["s_-5"] = AirP["Passengers"].shift(5)

# plot Results
fig, ax = plt.subplots(figsize=(10, 5))

AirP.plot(y=["Passengers", "s_4", "s_-5"], ax=ax)
plt.legend(["Original", "Backward shift", "Forward shift"])
plt.show()
```



Log-Returns

- can be interpreted as relative change in regards to previous time step

- Back-shift operator is applied when differences of time series are computed, since

$$x_i - x_{i-1} = x_i - B(x_i)$$

- Differencing is often combined with Box-Cox transformations
  - Example: **log-returns** of a (financial) time series are defined as

$$y_i = \log(x_i) - \log(x_{i-1}) = \log\left(\frac{x_i}{x_{i-1}}\right) = \log\left(\frac{x_i - x_{i-1}}{x_{i-1}} + 1\right) \approx \frac{x_i - x_{i-1}}{x_{i-1}}$$

  - Log-return y_i approximates the relative increase of the time series x_i at each time instance
- Last equation: Taylor series expansion of the logarithm:

$$log(s+1) = s - s^2/2 + ...$$

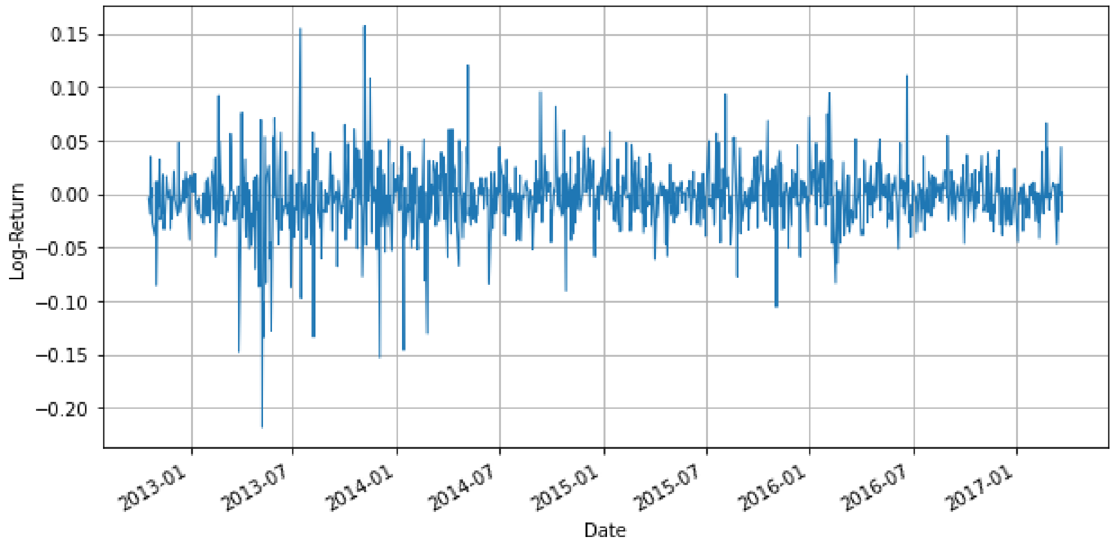
  - used in financials a lot

Code example log-returns

[120]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load data
tesla_stock = pd.read_csv(path_data+'Tesla.csv', parse_dates=True, decimal=",",␣
 ↪sep="\t")
# Create pandas DateTimeIndex
dtindex = pd.DatetimeIndex(data=tesla_stock["Date"], freq='infer')
# Set as Index
tesla_stock.set_index(dtindex, inplace=True)
tesla_stock.drop("Date", axis=1, inplace=True)

print(tesla_stock.head())

tesla_log_return = np.log(tesla_stock["Close"]) - np.log(tesla_stock["Close"].
 ↪shift(-1))
fig, ax = plt.subplots(figsize=(10, 5))
tesla_log_return.plot(ax=ax, linewidth=1)
ax.set_xlabel("Date")
ax.set_ylabel("Log-Return")
ax.grid()

plt.show()
```

```
            Close   Volume
Date
2012-10-19  27.74  1027302
```

```
2012-10-22  27.85   470198
2012-10-23  28.39   748998
2012-10-24  27.42  1016368
2012-10-25  27.52   577686
```



10.2 Decomposition of time series

- Time Series are usually dominated by a **trend** and/or **seasonal effects**
- a simple **additive decomposition** model is given by:

$$x_k = m_k + s_k + z_k$$

- where
  - k time index
  - x_k observed time series
  - m_k trend
  - s_k seasonal effect
  - z_k error term that is a sequence of correlated random variables with mean zero
    - not to mistake with noise. Noise is indipendent. This remainder term has correlation
- if seasonal effects increase as the trend increases, a **multiplicative** model may be more appropriate

$$x_k = m_k \cdot s_k + z_k$$

- if the noise is multiplicative as well, i.e. $x_k = m_k \cdot s_k \cdot z_k$, the logarithm of x_k is a linear model again

$$log(x_k) = log(m_k) + log(s_k) + log(z_k)$$

10.2.1 Moving Average

- for estimating the **trend** m_k and the **seasonal effect** s_k by means of the moving average filter:
  - Assume that $\{x_1, x_2, \dots, x_n\}$ is a time series and that $p \in \mathbb{N}$. The moving average filter of length p is defined as follows
    - If p is odd, then $p = 2l + 1$
      - meaning there are equally many elements to the left and right. So averaging all with $p = 2l + 1$
    - and the filtered sequence is defined by

$$g(x_i) = \frac{1}{p}\left(x_{i-l} + \cdots + x_i + \cdots + x_{i+l}\right)$$

    - If p is even, then $p = 2I$ and the filtered sequence is defined by

$$g(x_i) = \frac{1}{p}\left(\frac{1}{2}x_{i-l} + x_{i-l+1} + \cdots + x_i + \cdots + x_{i+l-1} + \frac{1}{2}x_{i+l}\right)$$

    - The value p is referred to as window width.
- **Moving average filter** amounts to replace the ith value in the time series by the average of the p nearest neighbors of x_i
- p
  - odd: then window stretches symmetrically aroung x_i
  - even: construction of window of length $p+1$ (which is then odd) but counts the endpoints by half
    - Example:
      - Monthly Data (frequency = 12) then
      - Trend component can be estimated by moving average with $p = 12$
      - Since even: January and December are weighted with 1/2
- since we average at each point over one period exactly, the seasonal effect vanishes and only the trend remains
  - yields the estimator $\hat{m}_k$

10.2.2 Seasonal effect

- To estimate the seasonal additive effect one computes

$$\hat{s}_k = x_k - \hat{m}_k$$

- Now the time series is $\hat{s}_k$ is averaged for each time point in one cycle
- We obtain a single estimate of the effect for each cycle point

10.2.3 Remainder Effect

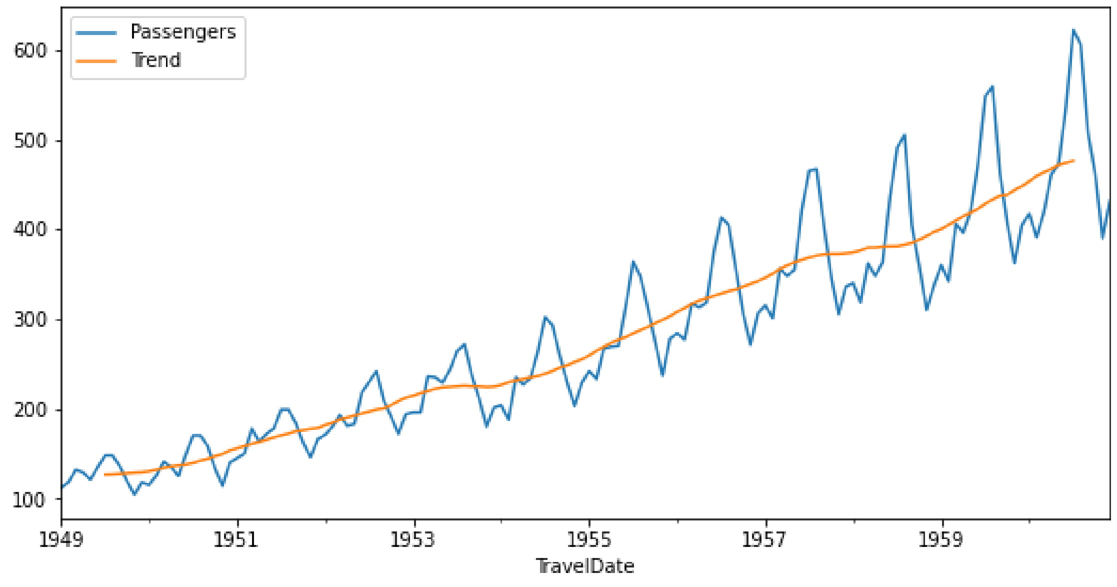
- We subtract the trend and seasonality estimates and arrive at the estimate for the remainder term:

$$\hat{r}_i = x_i - \hat{m}_i - \hat{s}_i$$

- The remainder term should consist of (possibly correlated) random values without structure / periodicity

10.2.4 Code Example Moving Average Trend Seasonal Effect Remainder

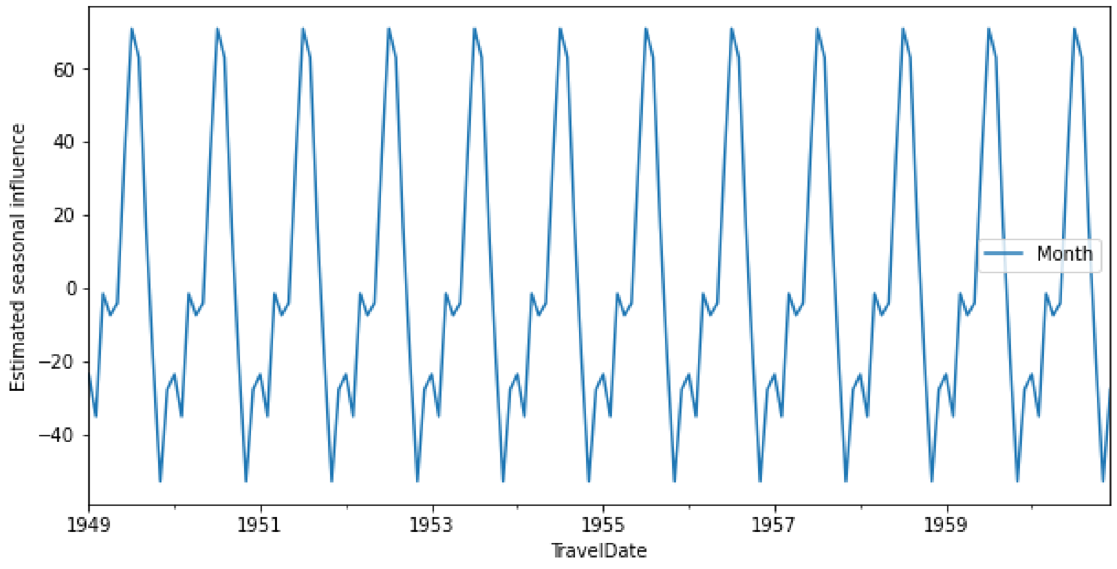
[121]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load data
AirP = pd.read_csv(path_data+'AirPassengers.csv', parse_dates=True)

# Create pandas DateTimeIndex
dtindex = pd.DatetimeIndex(data=pd.to_datetime(AirP["TravelDate"]),
                           freq='infer')
# Set as Index
AirP.set_index(dtindex, inplace=True)
AirP.drop("TravelDate", axis=1, inplace=True)

# Calculate and save Trend
AirP["Trend"] = AirP["Passengers"].rolling(window=12, center=True).mean()

# Plots
fig, ax = plt.subplots(figsize=(10, 5))
AirP.plot(y=["Passengers", "Trend"], ax=ax)
plt.show()
```



The estimated trend $\hat{m}_k$ line does not exhibit any seasonal fluctuations.

We take a look at the `AirPassengers` data and subtract the estimated trend and average over the months. The average per month is found by going over every month in the dataset, selecting all entries in the same month, and taking the mean value.

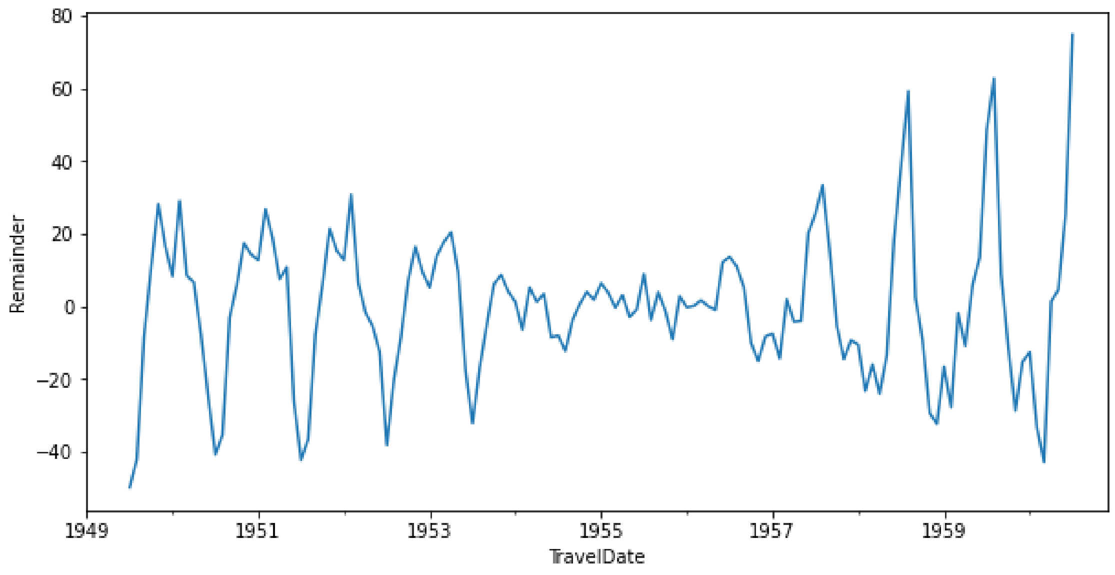
[122]:
```
# Create new Column Season, where the Trend is substracted
AirP["Season"] = AirP["Passengers"] - AirP["Trend"]

# Now find the mean per month
AirP["Month"] = np.zeros(AirP.shape[0])

# For every month, calculate mean:
for month in AirP.index.month.unique():
    mean = AirP.loc[AirP.index.month == month, "Season"].mean()
    AirP.loc[AirP.index.month == month, "Month"] = mean

# Plots
fig, ax = plt.subplots(figsize=(10, 5))
AirP.plot(y="Month", ax=ax)
ax.set_ylabel("Estimated seasonal influence")
plt.show()
```



We again cosider the `AirPassengers` data and subtract the estimated trend and seasonal effects

[123]:
```
# Create new Column remainder, where both season and Trend are substracted
AirP["Remainder"] = AirP["Passengers"] - AirP["Trend"] - AirP["Month"]

# Plots
fig, ax = plt.subplots(figsize=(10, 5))
AirP["Remainder"] .plot(ax=ax)
```

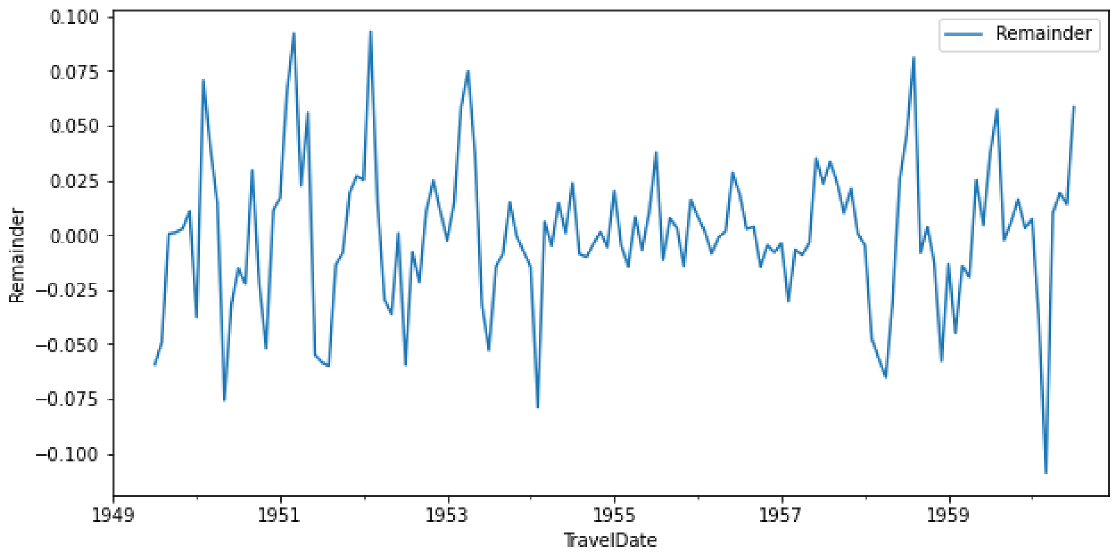
```
ax.set_ylabel("Remainder")
plt.show()
```



The figure shows the estimated remainder term $\hat{r}$. It is striking that there is non-random structure left in the remainder term. The reason for that is, that the linear decomposition model is not true in this case. We hence repeat the steps above for the *logarithm* of `AirPassengers` which amounts to a multiplicative model.

[124]:
```
# Manual Logarithmic transform
AirP_log = pd.DataFrame(
    data = {"log_Passengers": np.log(AirP["Passengers"])},
    index = AirP.index)

AirP_log["Trend"] = AirP_log["log_Passengers"]\
                    .rolling(window=12, center=True)\
                    .mean()
AirP_log["Season"] = AirP_log["log_Passengers"] - AirP_log["Trend"]
AirP_log["Month"] = np.zeros(AirP.shape[0])

# For every month, calculate mean:
for month in AirP_log.index.month.unique():
    mean = AirP_log.loc[AirP.index.month == month, "Season"].mean()
    AirP_log.loc[AirP_log.index.month == month, "Month"] = mean

AirP_log["Remainder"] = AirP_log["log_Passengers"] \
                        - AirP_log["Trend"] \
                        - AirP_log["Month"]
```

```
# Plots
fig, ax = plt.subplots(figsize=(10, 5))
AirP_log.plot(y="Remainder", ax=ax)
ax.set_ylabel("Remainder")
plt.show()
```



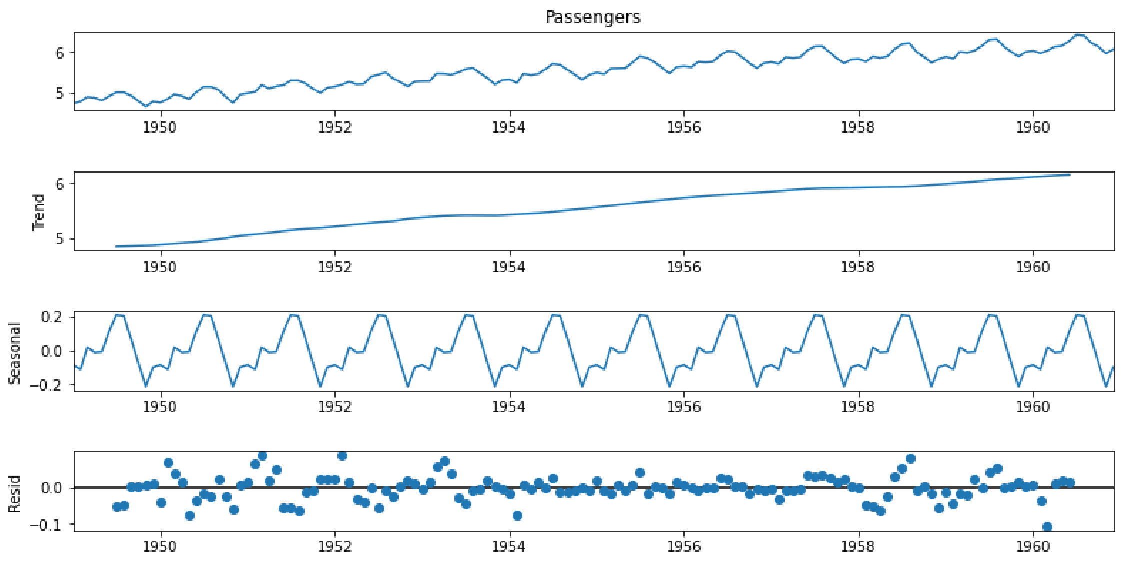
We see in the remainder estimated for the log-data that the non-random parts in the signal have diminished considerably.

10.2.5 Seasonal Decomposition of Time Series by Loess (STL)

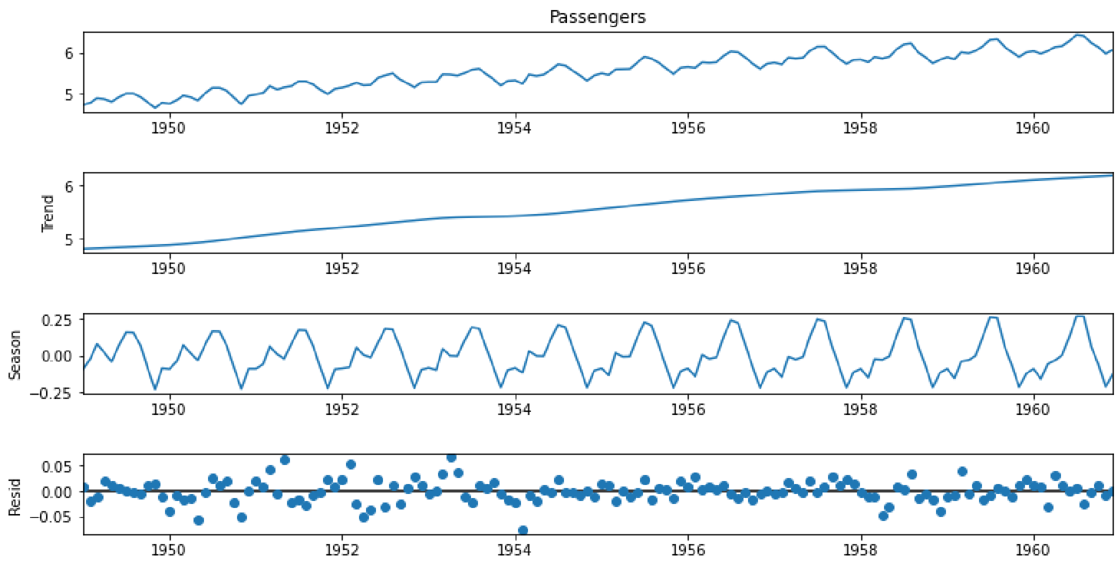
- Although the decomposition method gives promising results for our example data set, it is rarely used in practice, for several reasons:
  - Lacking robustness with respect to outliers in the data.
  - The seasonal component is assumed to be constant over time
- **State-of-the-art method** for decomposing time series that does not suffer from the above drawbacks is **seasonal decomposition of time series by loess (STL)**
- STL procedure is iterative : outliers in the estimated remainder terms are detected and their effect mitigated by proper reweighting
- Moving average smoothing is replaced by **loess regression** which gives more flexibility and better results as the moving average.
- Loess regression is a form of **local** (mostly linear) regression which means that the regression line through data $(x_1, y_1), ..., (x_n, y_n)$ at a point x is only computed using the observations in **a neighborhood** of x
- Seasonal component is not assumed to be constant : the method considers **cycle-subseries**, i.e. the subseries of values at each position of the seasonal cycle
- For example, for a monthly series with frequency 12, the first cycle-subseries consists of the January values, the second of the February values etc
- These sub-cycles are also smoothed by loess and may change over time

Code Example Seasonal Decomposition of Time Series by Loess (STL) There is a convenient way to perform a time series decomposition based on moving averages in **Python**. The `seasonal_decompose()` function from `statsmodels.tsa.seasonal` can be applied to a time series and performs the above steps. So the decomposition of the logarithmic `AirPassengers` data can be performed by the following code:

```
[125]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose

# Load data
AirP = pd.read_csv(path_data+'AirPassengers.csv', parse_dates=True)

# Create pandas DateTimeIndex
dtindex = pd.DatetimeIndex(data=pd.to_datetime(AirP["TravelDate"]),
                           freq='infer')
# Set as Index
AirP.set_index(dtindex, inplace=True)
AirP.drop("TravelDate", axis=1, inplace=True)

# Decomposition on log-model using seasonal_decompose()

decomp = seasonal_decompose(np.log(AirP["Passengers"]),
                            model = "additive")

# Plot
fig = decomp.plot()
fig.set_size_inches(12, 6)
plt.show()
```



[126]:
```python
#google colab may not have the correct version of statmodels. Install with
#!pip install statsmodels==0.12.1
from statsmodels.tsa.seasonal import STL

# Decomposition on log-model using STL
decomp = STL(np.log(AirP["Passengers"]), seasonal=9)
decomp = decomp.fit()

# Plot
fig = decomp.plot()
fig.set_size_inches(12, 6)
plt.show()
```



The figure shows the decomposition of the data. Note that the seasonal component is changing over time and the remainder term is much smaller than in the previous example.

- It becomes obvious that the seasonal contribution of air passengers in the winter months is decreasing. For instance, if we look at the January subseries, then we observe that the loess curve is decreasing. In other words, the seasonal effects for February are decreasing along the time axis.
- It becomes obvious that the seasonal contribution of air passengers in the summer months is increasing. For instance, if we look at the August subseries, then we observe that the loess curve is increasing. In other words, the seasonal effects for August are increasing along the time axis.

10.3 Mathmatical Concepts of Time Series

- Describing characteristics of data that seemingly fluctuate in a random fashion over time: we assume a time series can be considered as the **realization of random variables** that are **indexed over time**

10.3.1 Time series and Discrete Stochastic Processes

- Let T be a set of equidistant time points $T = \{t_1, t_2, ...\}$.
  1. A discrete stochastic process is a set of random variables $\{X_1, X_2, ...\}$. Each single random variable X_i has a univariate distribution function F_i and can be observed at time t_i.
  2. A time series $\{x_1, x_2, ...\}$ is a realization of a discrete stochastic process $\{X_1, X_2, ...\}$.
  - In other words, the value x_i is a realization of the random variable X_i measured at time t_i.
- Important distinction between a time series, i.e. a concrete observation of values, and a discrete stochastic process which is a theoretical construct to model the underlying mechanism that generates the values

10.3.2 Random Walk

- A person starts walking from the coordinate center with constant speed in x-direction. At each step, however, the person decides at random either to walk 1 m to the left or to the right.
- This is the simplest instance of a random walk
- Probabilistic model for this random walk
  1. Choose n independent Bernoulli random variables $D_1, \dots, D_n$ that take on the values 1 and -1 with equal probability, i.e. $p = 0.5$
  2. Define the random variables $X_i = D_1 + \dots + D_i$ for each $1 \leq i \leq n$. Then $\{X_1, X_2, ...\}$ is a discrete stochastic process modeling the random walk
- From the definition of the process it is clear that the following recursive definition is equivalent

$$X_i = X_{i-1} + D_i, \quad X_0 = 0.$$

- If in each step a fixed constant δ is added to the series, i.e.

$$Y_i = \delta + Y_{i-1} + D_i, \quad Y_0 = 0.$$

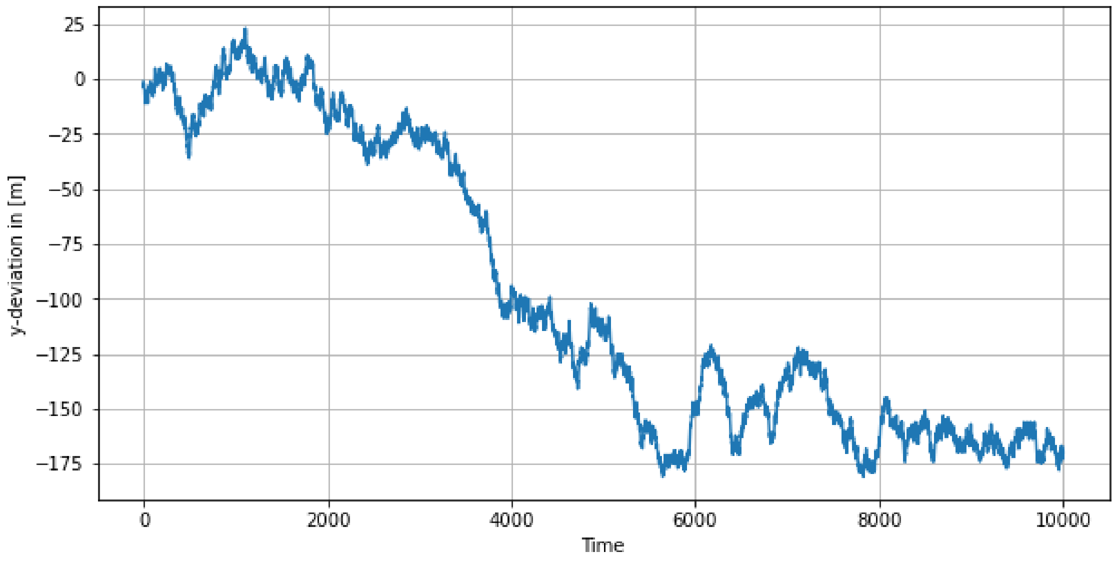
  then we obtain a random walk with drift
- The random walk with drift models are used to model trends in time series data

Code Example Random Walk Assume, that a person starts walking from the coordinate center with constant speed in x-direction. At each step, however, the person decides *at random* either to walk 1m to the left or to the right. This is the simplest instance of a *random walk*. The probabilistic model for this random walk would be

1. Choose n independent Bernoulli random variables $D_1, \dots, D_n$ that take on the values 1 and -1 with equal probability, i.e. $p = 0.5$.
2. Define the random variables $X_i = D_1 + \dots + D_i$ for each $1 \leq i \leq n$. Then $\{X_1, X_2, \dots\}$ is a discrete stochastic process modeling the random walk.

The following Python-code computes a particular instance of this process, i.e. a time series $\{x_1, x_2, \dots\}$. Each time this code is re-run, a new time series will display. The function `np.cumsum()` computes the cumulated sum of a given vector.

```
[127]: import matplotlib.pyplot as plt
       import numpy as np

       nsamp = 10000
       # Random samples d and cumulative sum x
       d = np.random.choice(a=[-1,1], size=nsamp, replace=True)

       x = np.cumsum(d)

       # Plot
       fig = plt.figure(figsize=(10, 5))
       ax = fig.add_subplot(1, 1, 1)
       ax.plot(x)
       ax.grid()
       plt.xlabel("Time")
       plt.ylabel("y-deviation in [m]")

       plt.show()
```



From the definition of the process it is clear that the following recursive definition is equivalent

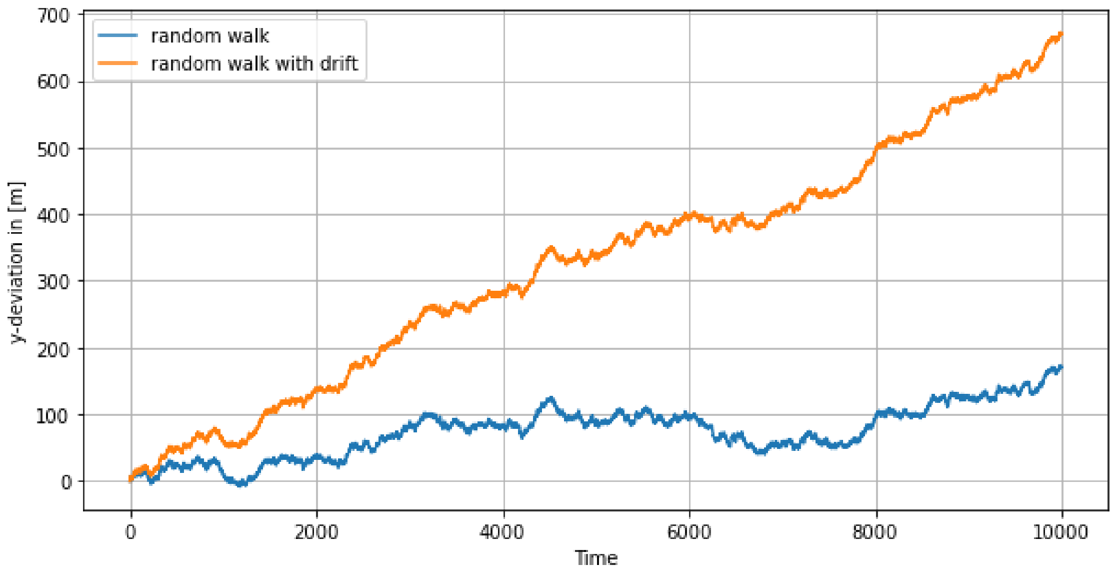
$$X_i = X_{i-1} + D_i, \quad X_0 = 0.$$

If in each step a fixed constant δ is added to the series, i.e.

$$Y_i = \delta + Y_{i-1} + D_i, \quad Y_0 = 0.$$

then we obtain a *random walk* with *drift*. In the figure, we see the observed time series of such a process. The random walk with drift models are used to model trends in time series data.

[128]:
```
# Set random seed
np.random.seed(0)
# Random samples d and cumulative sum x
d = np.random.choice(a=[-1,1], size=nsamp, replace=True)
x = np.cumsum(d)
# Add drift delta
delta = 5e-2
y = np.linspace(0, nsamp*delta, nsamp)
y += x

# Plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(x, label="random walk")
ax.plot(y, label="random walk with drift")
ax.grid()
plt.xlabel("Time")
plt.ylabel("y-deviation in [m]")
plt.legend()

plt.show()
```



10.3.3 Stochastic Process

- A time series

$$\{x_1, x_2, ..., x_n\}$$

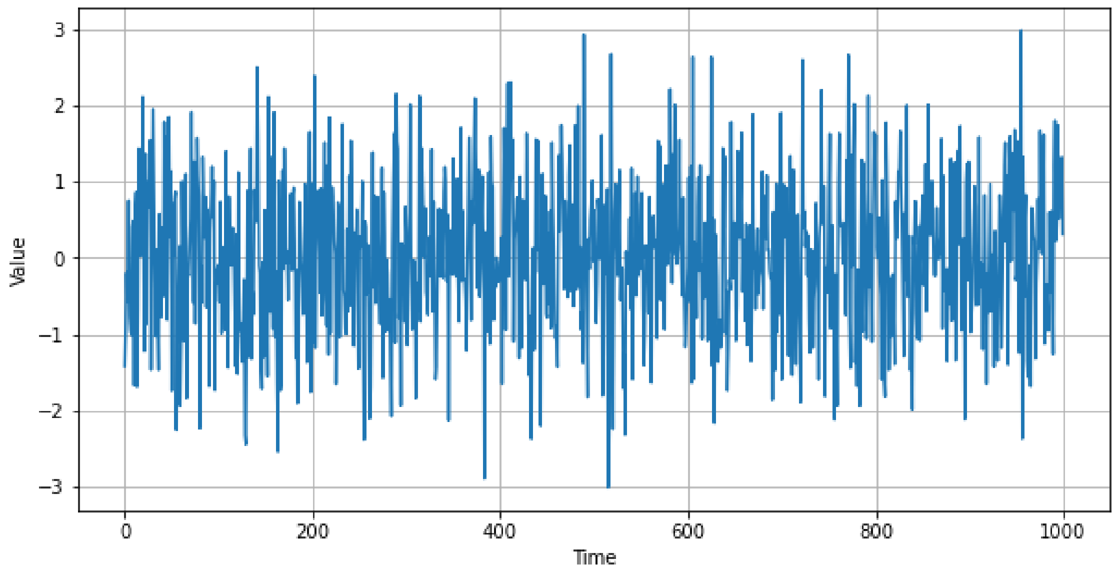
can be understood as **one** realization of a multivariate random variable

$$\{X_1, X_2, ..., X_n\}$$

- Modeling and prediction for time series hence amounts to analyze a data set with one observation, which is impossible without further assumptions on the series.
- Example that lacks all these assumptions and thus is impredictible: **white noise**
- A white noise process consists of independent and identically distributed random variables $\{W_1, W_2, ...\}$ where each W_i has mean 0 and variance σ^2
  - if in addition the individual random variables W_i are normally distributed $\rightarrow$ **Gaussian white noiese**
  - there models describe **noise** in engineering applications
  - the observations of a white noise process are **uncorrelated** and could hence be treated with ordinary statistical methods

Code Example White Noise process Stochastic Process A white noise process consists of independent and identically distributed random variables $\{W_1, W_2, \dots\}$ where each W_i has mean 0 and variance σ^2.

[129]:
```python
import matplotlib.pyplot as plt
import numpy as np

# White noise signal
w = np.random.normal(size=1000)

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(w)
ax.grid()
plt.xlabel("Time")
plt.ylabel("Value")

plt.show()
```



If in addition the individual random variables W_i are normally distributed, then the process is called *Gaussian* white noise. These models are used to describe noise in engineering applications. The term *white* is chosen in analogy to white light and indicates that all possible periodic oscillations are present in a time series originating from a white noise process with equal strength.

10.3.4 Discrete Stochastic Processes Generated from White Noise

- We apply a sliding window filter to the white noise process

$$\{W_1, W_2, \ldots\}$$

- We obtain a **moving average process**
- If we choose the window length to be 3 , we obtain

$$V_i = \frac{1}{3}\left(W_{i-1} + W_i + W_{i+1}\right)$$

- We choose

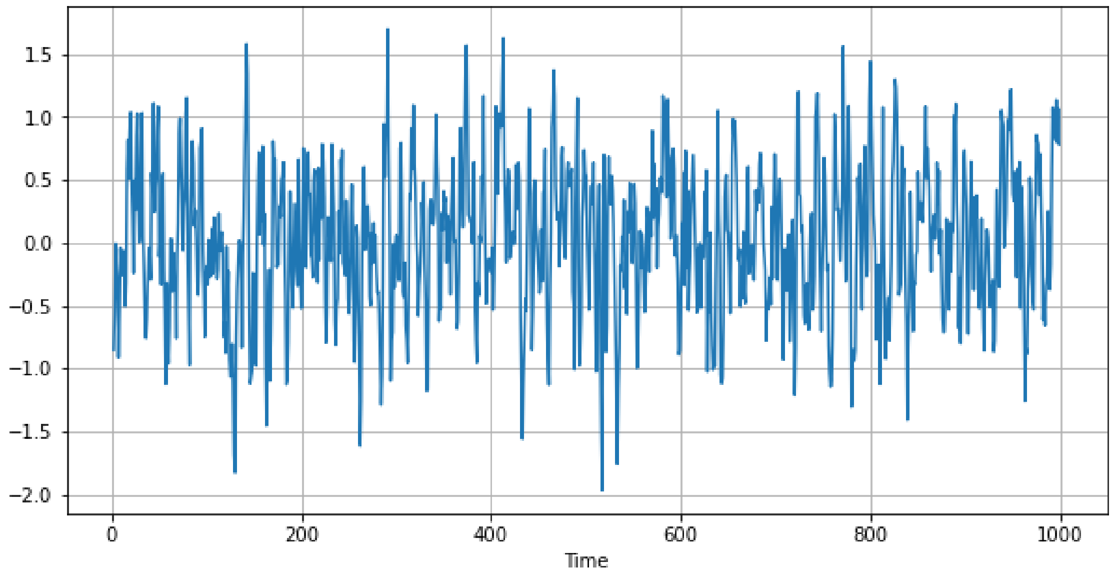
$$V_1 = W_1 \quad \text{and} \quad V_2 = 0.5\left(W_1 + W_2\right)$$

Code Example Moving Average Process on White Noise If we apply a sliding window filter to the white noise process $\{W_1, W_2, \ldots\}$ in the previous example of white noise we obtain a *moving average* process. E.g. if we choose the window length to be 3, we obtain

$$V_i = \frac{1}{3}(W_{i-1} + W_i + W_{i+1}).$$

We choose $V_1 = W_1$ and $V_2 = 0.5(W_1 + W_2)$. The resulting process is smoother, i.e. the higher order oscillations are smoothed out. In `Python` a sliding window can be achieved using `.rolling()`, the mean can be found with `.mean()`.

```
[130]: import pandas as pd

       # Convert to DataFrame
       w = pd.DataFrame(w)
       # Filter with window = 3
       v = w.rolling(window=3).mean()

       # plot
       fig = plt.figure(figsize=(10, 5))
       ax = fig.add_subplot(1, 1, 1)
       ax.plot(v)
       ax.grid()
       plt.xlabel("Time")

       plt.show()
```



10.3.5 Autoregressive Time Series

- the concept of autoregressive processes of order p (AR(p)-processes). Intuitively, the value of an AR(p) processes at time n is a *linear combination* of the p previous values in the series. To be more precise
- Many examples of real world time series, e.g. acoustic time series in speech analysis, contain dominant oscillatory components, producing a sinusoidal type of behaviour
- One possible model to generate such quasi-periodic data are autoregressive series.
- In other words, the value of the process at time instance i is modelled as a linear combination of the past two (or p) values plus some random component. Therefore this process is called

autoregressive

- In the definition of a discrete stochastic process $\{Xl, X_2, ...\}$ we have postulated the existence of a distribution function $F_i(x)$ for each observation X_i in the process, i.e.

$$P(X_i \leq x) = F_i(x)$$

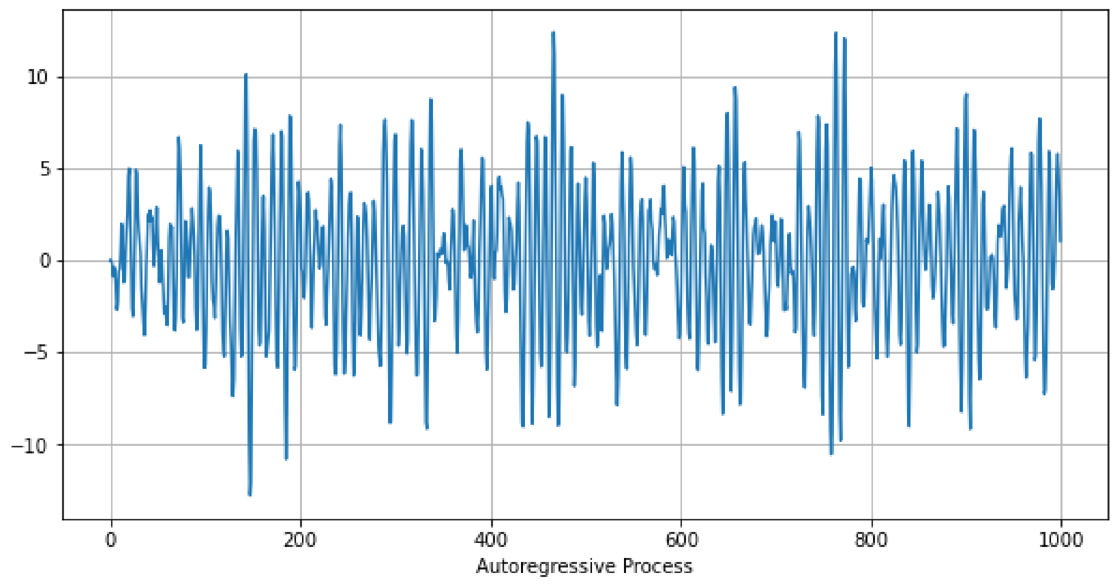
- Knowledge of individual distributions, however, is not sufficient to understand the **serial** behaviour of the process, because the observations are mutually **dependent**

Code Example Autoregressive Time Series We consider again the white noise process $\{W_1, W_2, ...\}$ and recursively compute the following sequence

$$X_i = 1.5X_{i-1} - 0.9X_{i-2} + W_i.$$

In other words, the value of the process at time instance i is modelled as a linear combination of the past two values plus some random component. Therefore this process is called *autoregressive*. The definition of the initial conditions is subtle, because the process will strongly depend on these. We will for the time being ignore the issue of initial conditions.

[131]:
```
# Autoregressive filter:
ar = np.zeros(1000)
for i in range(2,1000):
    ar[i] = 1.5 * ar[i-1] - 0.9 * ar[i-2] + w.iloc[i]

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(ar)
ax.grid()
plt.xlabel("Autoregressive Process")

plt.show()
```



The figure shows a realization of the autoregressive process above. The oscillatory behaviour becomes clearly visible.

- Another interpretation of the autoregressive process above is via differential equations.
- The finite difference scheme for the second order equation

$$\ddot{x} + 2\delta\dot{x} + \omega_0^2 x = W(t)$$

  is given by

$$\frac{x_{i-2} - 2x_{i-1} + x_i}{\Delta t^2} + 2\delta\frac{x_i - x_{i-1}}{\Delta t} + \omega_0^2 x_i = W_i$$

- Here δ is the damping term and ω_0 the frequency of the homogeneous equation.
- Setting $\Delta t = 1$, $\omega_0^2 = 0.4$ and $\delta = 0.05$ gives – after some rearrangements – the autoregressive process above.
- Hence it can be seen as a **harmonic oscillator with random input**.
- The wave length of the exact solution is

$$T = \frac{2\pi}{\omega_0} \approx 10.0$$

  which matches the previous situation pretty well.

Measures of dependence

- Aside to the individual (also called **marginal**) distributions F_i of the random variables in the process we define **first** and **second order** moments to analyse the whole process.
- We start with the first order moments of the process, the mean sequence:
- The mean sequence $\{\mu(1), \mu(2), ...\}$ (or mean function) of a discrete stochastic process $\{X_1, X_2, ...\}$ is defined as the sequence of the means:

$$\mu(i) = \mathrm{E}[X_i]$$

Example

- If W_i denotes a white noise process, then $\mathrm{E}[X_i] = 0$ for all $i \geq 1$.
- Averaging the values in the process does not change the mean and the mean sequence of a moving average process, and hence is also 0 .
- If X_i is a random walk with drift, i.e. $X_0 = 0$ and

$$X_i = \delta + X_{i-1} + W_i$$

  then we find that

$$\begin{aligned}\mathrm{E}[X_1] &= \delta + \mathrm{E}[X_0] + \mathrm{E}[W_1] = \delta\\ \mathrm{E}[X_2] &= \delta + \mathrm{E}[X_1] + \mathrm{E}[W_2] = 2\delta\\ \mathrm{E}[X_3] &= \delta + \mathrm{E}[X_2] + \mathrm{E}[W_3] = 3\delta\\ &\vdots\end{aligned}$$

- This means: $\mu(i) = i\delta$

Autocovariance and autocorrelation

- The autocovariance measures the linear dependence of two points on the same process observed at different times
- If a series is very smooth, the autocovariance is large, even if i and j are far apart
- Autocorrelation hence gives a rough measure how well the series at time i can be forecast by the value at time j
- as second order moments, we consider the covariance within a *single* process
- Let $\{X_1, X_2, ...\}$ be a discrete stochastic process.
  1. The autocovariance γx is defined as
  $$\gamma_X(i,j) = \operatorname{Cov}\left(X_i, X_j\right) = \operatorname{E}\left[\left(X_i - \mu(i)\right)\left(X_j - \mu(j)\right)\right]$$
  2. The autocorrelation ρ_X is defined as
  $$\rho_X(i,j) = \frac{\gamma_X(i,j)}{\sqrt{\gamma_X(i,i)\gamma_X(j,j)}}.$$
- For $i = j$ the autocovariance reduces to the **variance** of X_i
- Important property of autocovariance and autocorrelation: they are symmetric, i.e.
$$\gamma(i,j) = \gamma(j,i)$$
- Note that if
$$\gamma(i,j) = 0$$
this only means that X_i and X_j are not linearly dependent, however, they still may depend in some nonlinear way
- The **autocorrelation** is the normalized version of autocovariance, i.e.
$$\rho(i,j) \in [-1,1]$$
if there is a linear relationship between X_i and X_j, then
$$\rho\left(X_i, Y_j\right) = \pm 1$$
- To be more precise : if
$$X_i = \beta_0 + \beta_1 X_j$$
the correlation will be 1 if $\beta_1 > 0$, and -1 else.
- Autocorrelation hence gives a rough measure how well the series at time i can be forecast by the value at time j

Calculation example: Autocovariance and Autocorrelation

- A white noise process has the autocovariance function

$$\gamma(i,j) = \begin{cases} 0 & \text{if } i \neq j \\ \sigma^2 & \text{if } i = j \end{cases}$$

- Accordingly, the autocorrelation is 1 if $i = j$ and 0 else
- Autocovariance of the **three point moving average process**
- From the properties of the covariance it becomes clear that

$$\gamma(i,j) = \mathrm{Cov}\left(X_i, X_j\right) = \mathrm{Cov}\left(\frac{1}{3}\left(W_{i-1} + W_i + W_{i+1}\right), \frac{1}{3}\left(W_{j-1} + W_j + W_{j+1}\right)\right)$$

- If $i = j$, then

$$\begin{aligned} \mathrm{Cov}\left(X_i, X_i\right) &= \frac{1}{9}\mathrm{Cov}\left(W_{i-1} + W_i + W_{i+1}, W_{i-1} + W_i + W_{i+1}\right) \\ &= \frac{1}{9}\left(\mathrm{Cov}\left(W_{i-1}, W_{i-1}\right) + \mathrm{Cov}\left(W_i, W_i\right) + \mathrm{Cov}\left(W_{i+1}, W_{i+1}\right)\right) \\ &= \frac{3\sigma^2}{9} \end{aligned}$$

- This follows from the fact, that W_i, W_{i-1} and W_{i+1} are mutually uncorrelated
- Analogously for $i + 1 = j$:

$$\begin{aligned} \mathrm{Cov}\left(X_i, X_{i+1}\right) &= \frac{1}{9}\mathrm{Cov}\left(W_{i-1} + W_i + W_{i+1}, W_i + W_{i+1} + W_{i+2}\right) \\ &= \frac{1}{9}\left(\mathrm{Cov}\left(W_i, W_i\right) + \mathrm{Cov}\left(W_{i+1}, W_{i+1}\right)\right) \\ &= \frac{2\sigma^2}{9} \end{aligned}$$

- To summarize, we find

$$\gamma(i,j) = \begin{cases} \frac{3\sigma^2}{9} & \text{if } i = j \\ \frac{2\sigma^2}{9} & \text{if } |i-j| = 1 \\ \frac{\sigma^2}{9} & \text{if } |i-j| = 2 \\ 0 & \text{else} \end{cases}$$

- Smoothing of the white noise introduces a **nontrivial autocovariance structure**
- Noteworthy: autocovariance only depends on the distance of the observations, but not on their value.

10.3.6 Stationarity

- A time series can be seen as a single realization of a multivariate random variable $\{X1, X2, ...\}$ which we call in this special setting a stochastic process
- As we already know from elementary statistics, there is no way to do sound statistical analysis on a single observation
- New Concept of regularity that allows us to infer information from a single time series: $\rightarrow$ stationarity

Strict stationarity A discrete stochastic process is called strictly stationary if for each finite collection $\left\{X_{i_1}, \ldots, X_{i_n}\right\}$ and each lag $h \in \mathbb{Z}$ the shifted collection

$$\left\{X_{i_1+h}, \ldots, X_{i_n+h}\right\}$$

has the same distribution. Put differently:

$$P\left(X_{i_1} \leq c_1, \ldots, X_{i_n} \leq c_n\right) = P\left(X_{i_1+h} \leq c_1, \ldots, X_{i_n+h} \leq c_n\right)$$

for any $c_1, \ldots, c_n$.

- In other words : the probabilistic character of the process does not change over time
- This amounts to say that the marginal distributions $F_i = F$ of the process coincide and in particular this implies that
$$\mu x(i) = \mu x(j)$$
- the **mean sequence is constant**

Weak Stationarity

- These two conclusions from strict stationarity are sufficient for most applications in order to come up with reasonable statistical models.
- Hence they give rise to the definition of a weaker form of stationarity
- A discrete stochastic process X_i is called weakly stationary if
  1. the mean sequence $\mu_X(i)$ is constant and does not depend on the time index i and
  2. the autocovariance sequence $\gamma_X(i,j)$ depends on i and j only through their difference $|i-j|$.

Calculation Example Weak Stationarity

- Reconsider the three point moving average process
- It is obvious that the mean function
$$\mu(i) = \mu = 0$$
is constant
- The autocovariance only depends on the time lags
$$\gamma(h) = \begin{cases} \frac{3\sigma^2}{9} & \text{if } h = 0 \\ \frac{2\sigma^2}{9} & \text{if } |h| = 1 \\ \frac{\sigma^2}{9} & \text{if } |h| = 2 \\ 0 & \text{else.} \end{cases}$$
- Moving average process thus is weakly stationary

Testing Stationarity

- Stationarity is a property of stochastic processes
- **Example**: time series decomposition where trend and seasonality of a time series are estimated and subtracted which yields the so called **remainder sequence**

Tests * The first and simplest type of test is to **plot** the time series and look for evidence of trend in mean sequence or in the autocorrelation function * If any such patterns are present then these are **signs of non-stationarity** * compute the mean and autocovariance sequences seperately for several windows and compare their behaviour * When there is a dramatic change, no stationarity

Estimation of mean sequence

- Serial dependence measure, such as autocovariance, are restricted to stochastic processes and hence make use of the probabilistic behaviour at each time instance
- In practice, however, we are confronted with a single realization of the process, i.e. at each time instance only one value is available
- Due to the stationarity assumption of the process we know that the mean sequence $(k) =$ is constant. A canonical estimator hence is

$$\hat{\mu} = \overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Estimation of autocovariance

- The theorectical autocovariance function is estimated by the sample autocovariance sequence:
- Sample autocovariance
  1. The sample autocovariance is defined by

     $$\hat{\gamma}(h) = \frac{1}{n}\sum_{i=1}^{n-h}\left(x_{i+h} - \bar{x}\right)\left(x_i - \bar{x}\right)$$

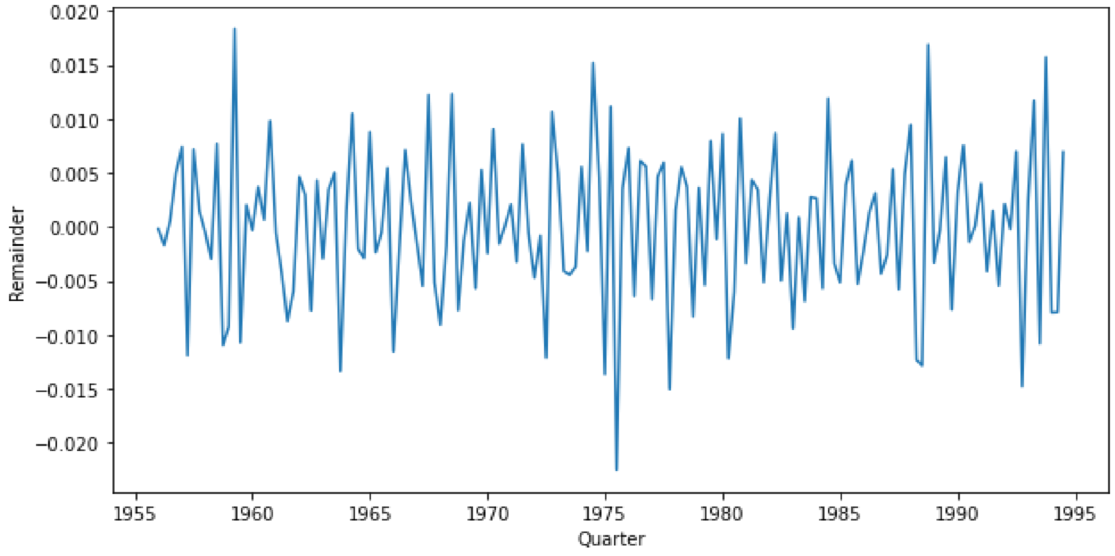
     with $\hat{\gamma}(-h) = \hat{\gamma}(h)$ for $h = 0, 1, \dots, n-1$
  2. The sample autocorrelation is defined by

     $$\hat{\rho}(h) = \frac{\hat{\gamma}(h)}{\hat{\gamma}(0)}.$$

Code Example Testing Stationarity We study the STL decomposition of two time series from Chapter Introduction to Time Series. We begin with the quarterly Australian electricity production. We read the data into **Python** and decompose the logarithm of the data into a trend, a seasonal component and a remainder. We just plot the remainder of the series.

[132]:
```
#!pip install statsmodels==0.12.1
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from statsmodels.tsa.seasonal import STL

# Load data
```

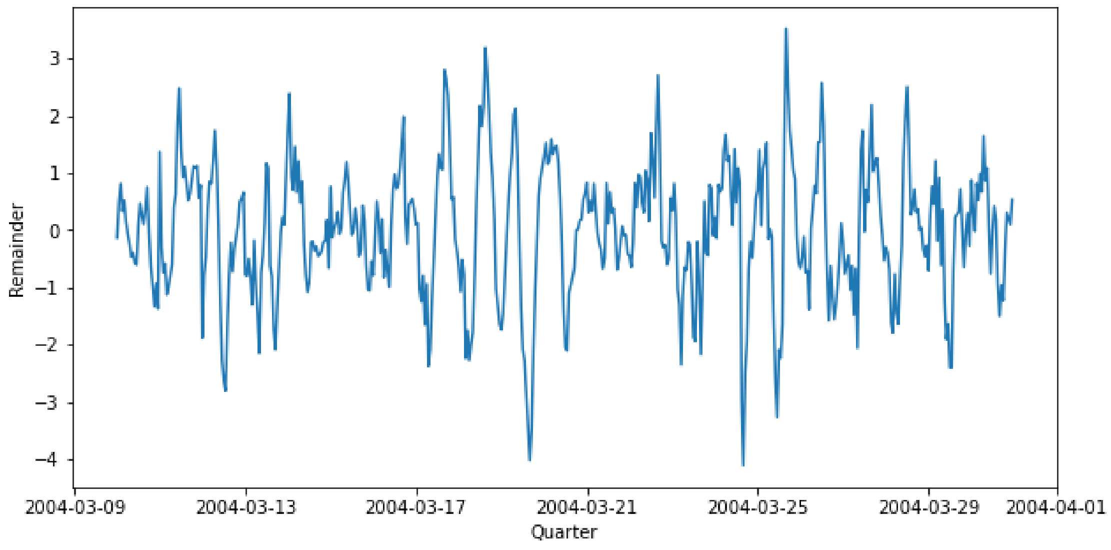
```
AusEl = pd.read_csv(path_data+'AustralianElectricity.csv', sep = ";")
# Create pandas DateTimeIndex
dtindexE = pd.DatetimeIndex(data=pd.to_datetime(AusEl["Quarter"]),
                            freq='infer')

# Set as Index
AusEl.set_index(dtindexE, inplace=True)
AusEl.drop("Quarter", axis=1, inplace=True)

# Decomposition on log-model using STL
decomp = STL(np.log(AusEl), seasonal=15)
decomp = decomp.fit()

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(decomp.resid)
plt.xlabel("Quarter")
plt.ylabel("Remainder")

plt.show()
```



The series does not exhibit any seasonal patterns, has a constant mean (0) and roughly constant variance. From visual inspection one would conclude that the underlying process is stationary.

The next example is the air temperature measurement. We again read the data, choose a window of appropriate size and coerce it into a time series. Afterwards we decompose the data (without log transform) by STL and study the remainder sequence.

[133]:
```python
# Load data
AirQ = pd.read_csv(path_data+'AirQualityUCI.csv',
                   parse_dates=True, decimal=",", sep=';')

# Combine Date and Time Columns:
AirQ["Time"] = AirQ["Time"].str.replace(".", ":")
AirQ["Date"] = pd.to_datetime(AirQ["Date"] + " " + AirQ["Time"])
dtindex = pd.DatetimeIndex(data=pd.to_datetime(AirQ["Date"]),
                           freq='infer')

# Set as Index
AirQ.set_index(dtindex, inplace=True)
AirQ = AirQ.sort_index()
# Only keep temperature for given period
AirT = AirQ.loc["2004-3-10":"2004-3-30", "T"]
# AirT = pd.DataFrame(AirQ["T"])

# Decomposition on log-model using STL
decomp = STL(AirT, period=24 , seasonal=9)
decomp = decomp.fit()

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(decomp.resid)
plt.xlabel("Quarter")
plt.ylabel("Remainder")

plt.show()
```



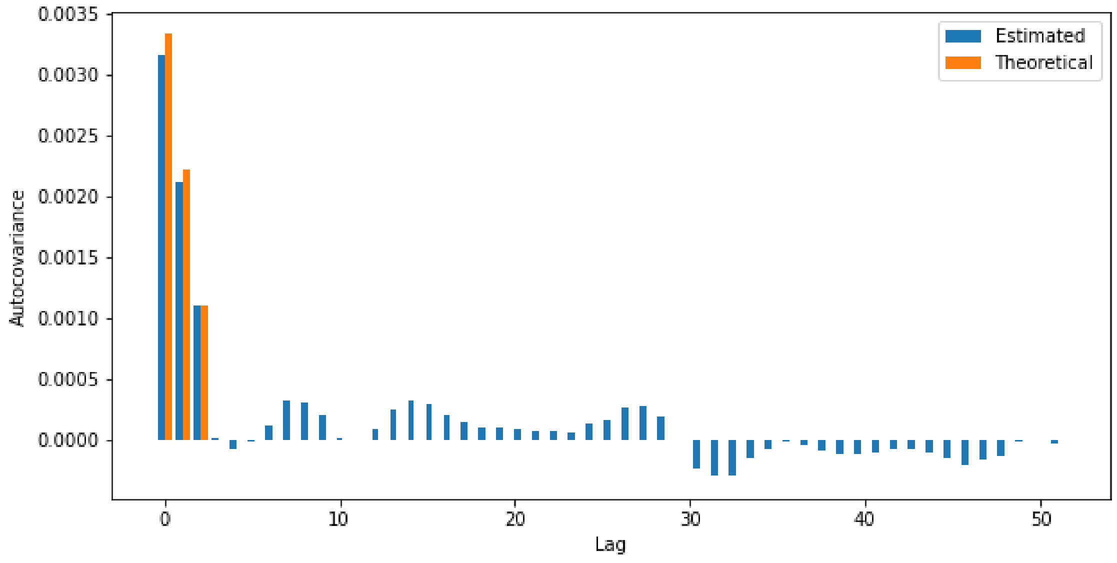
The time series plot still exhibits some seasonality, such that stationarity of the underlying process seems unlikely. Different choices of **seasonal** in the **STL()** function will change the behaviour of the remainder sequence.

Code Example Estimation of autocovariance We compute the estimators above for simulated data. First we simulate a realization of a moving average process.

[134]:
```
import numpy as np
import pandas as pd

np.random.seed(0)

# White noise signal
w = np.random.normal(size=1000, loc=2, scale=0.1)
w = pd.DataFrame(w)
# Filter with window = 3
w = w.rolling(window=3).mean()
```

In `Python`, the autocovariance and -correlation can be computed with the `acof()` and `acf()` (from `statsmodels.tsa.stattools`) respectively. During the application of the moving average (in the `rolling` command), missing values are created at the boundaries. To omit `NaN` entries, the setting `missing` is set to `drop`.

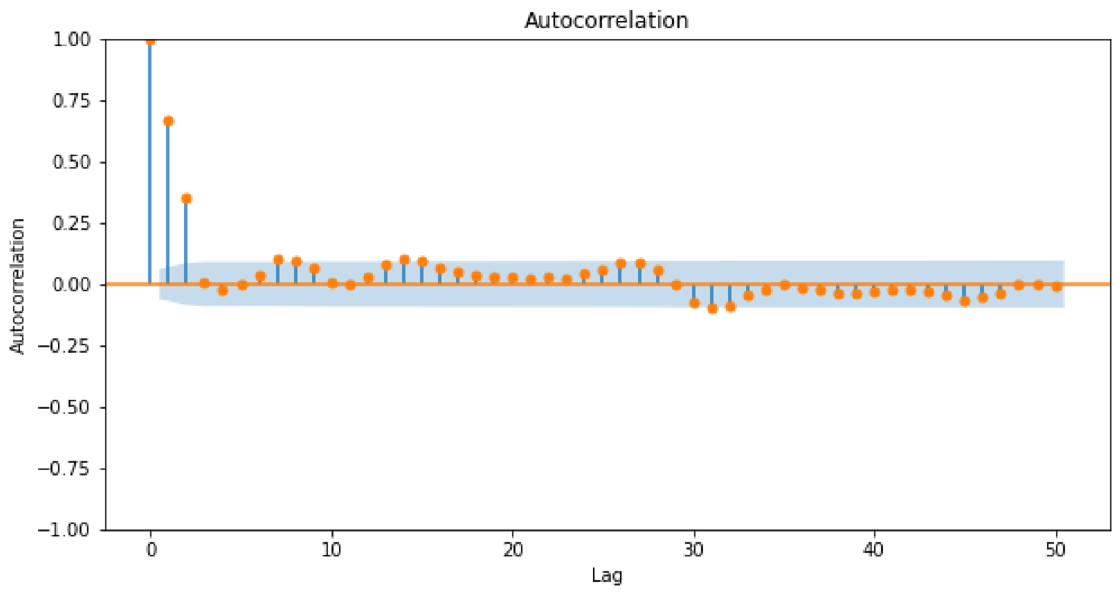
[135]:
```
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acovf

# Autocovariance
ac = acovf(w, fft=False, missing='drop', nlag=50)

# Theoretical autocovariance
sigma = 0.1
act = np.zeros(len(ac))
act[0] = 3 / 9 * sigma**2
act[1] = 2 / 9 * sigma**2
act[2] = 1 / 9 * sigma**2


# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)

width = 0.4
x = np.linspace(0, len(ac), len(ac))
ax.bar(x - width / 2, ac, width=width, label='Estimated')
ax.bar(x + width / 2, act, width=width, label='Theoretical')
plt.xlabel("Lag")
plt.ylabel("Autocovariance")
plt.legend()

plt.show()
```



The `nlag` parameter sets the maximal lag up to which the autocovariance should be computed. The figure shows the autocovariance of the moving average data. It can be seen that the lags $h = 0, 1, 2$ produce a significant value and that for $h > 2$ the autocovariance is very small. This is in accordance with the theoretical results. We add the values of the true autocovariance for comparison. Nevertheless, the sample autocovariance shows an oscillating pattern which we will comment on below.

We again consider the moving average process. the following command produces the correlogram.

[136]:
```
from statsmodels.graphics.tsaplots import plot_acf

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
plot_acf(w, missing='drop', lags=50, ax=ax, c='C1')
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.show()
```

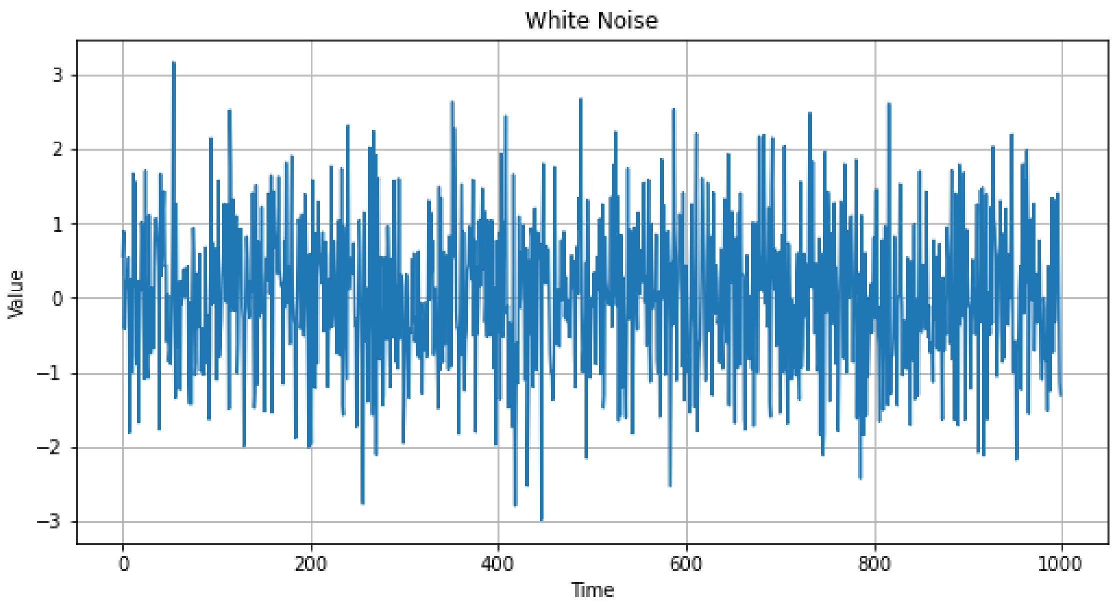


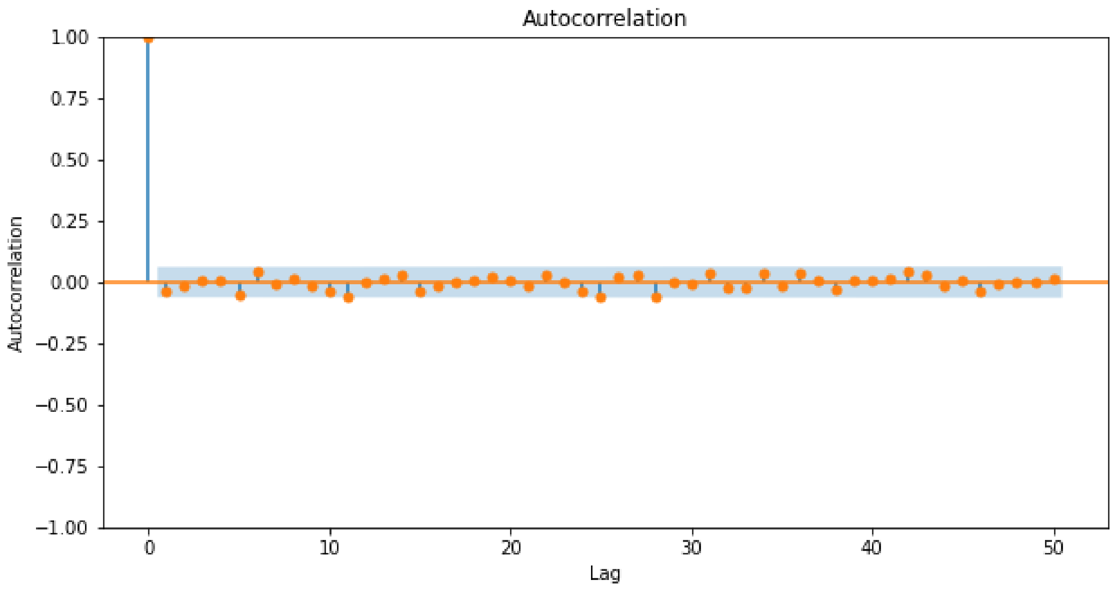
The two-sigma confidence bands are drawn automatically and we see that starting from the 3rd all lags produce a autocorrelation which is in the band. In practice, once the estimated autocorrelation stays within these bands one sets it to 0 (A more formal way of testing is the *Ljung-Box test* that tests the null-hypothesis that a number of autocorrelation coefficients is zero. The test is implemented in the `statsmodels.stats.diagnostic.acorr_ljungbox()`)

Code Example Autocovariance Function on White Noise, Random Walk, Moving Average and Auto Regressive White Noise

[137]:
```
import matplotlib.pyplot as plt
import numpy as np

# White noise signal
w = np.random.normal(size=1000)

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(w)
ax.grid()
plt.xlabel("Time")
plt.ylabel("Value")
plt.title("White Noise")

plt.show()

fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
```

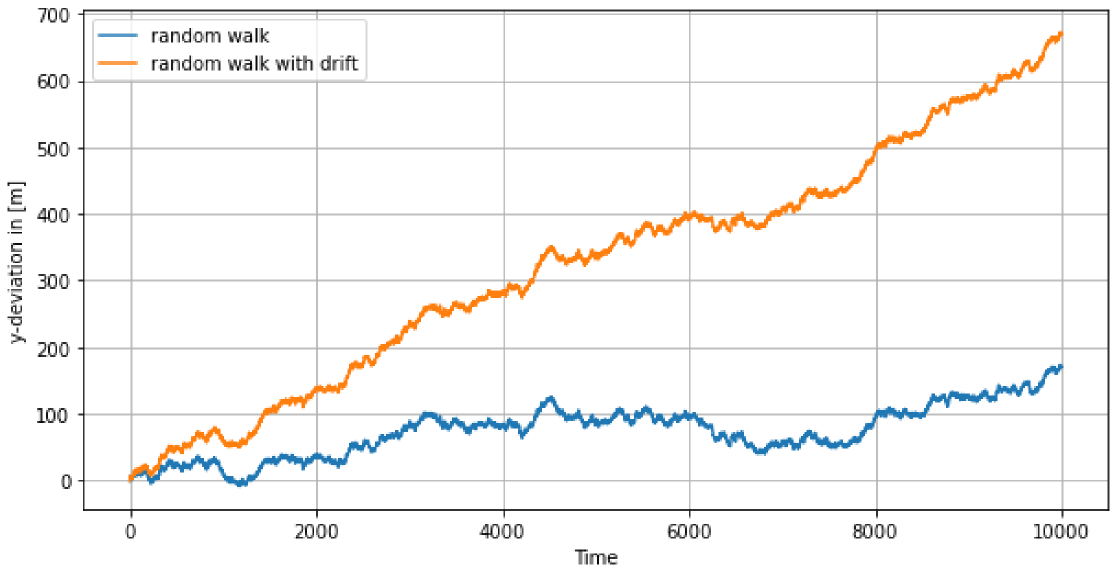
```
plot_acf(w, missing='drop', lags=50, ax=ax, c='C1')
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.show()
```



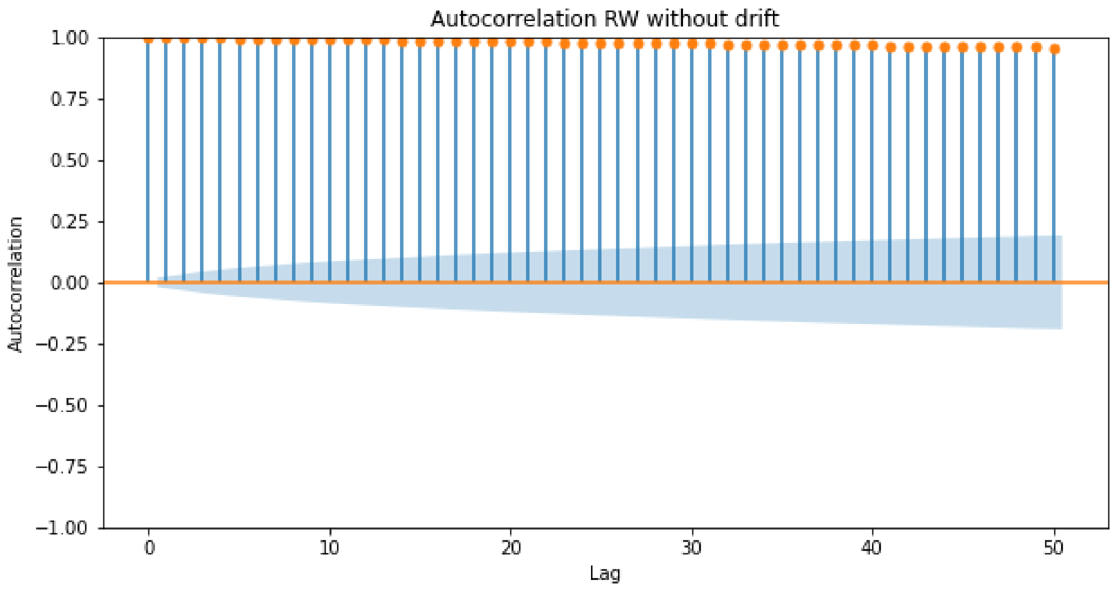


Random Walk

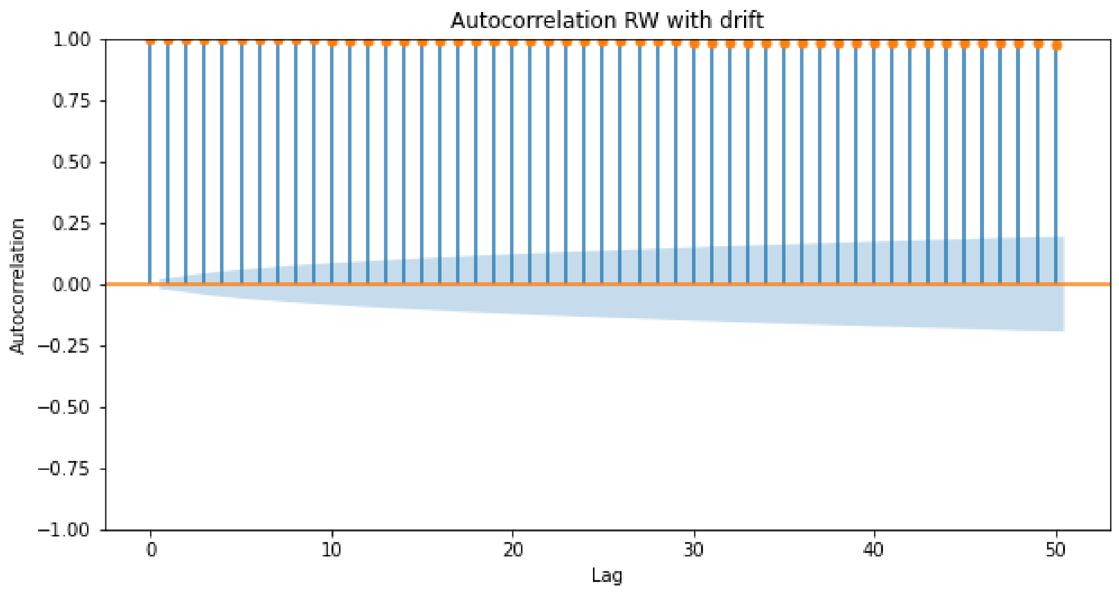
[138]:
```
# Set random seed
np.random.seed(0)
# Random samples d and cumulative sum x
nsamp = 10000
d = np.random.choice(a=[-1,1], size=nsamp, replace=True)
x = np.cumsum(d)
# Add drift delta
delta = 5e-2
y = np.linspace(0, nsamp*delta, nsamp)
y += x

# Plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(x, label="random walk")
ax.plot(y, label="random walk with drift")
ax.grid()
plt.xlabel("Time")
plt.ylabel("y-deviation in [m]")
plt.legend()

plt.show()

fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
plot_acf(x, missing='drop', lags=50, ax=ax, c='C1')
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.title("Autocorrelation RW without drift")
plt.show()

fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
plot_acf(y, missing='drop', lags=50, ax=ax, c='C1')
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.title("Autocorrelation RW with drift")
plt.show()
```



random walk
random walk with drift
y-deviation in [m]
Time

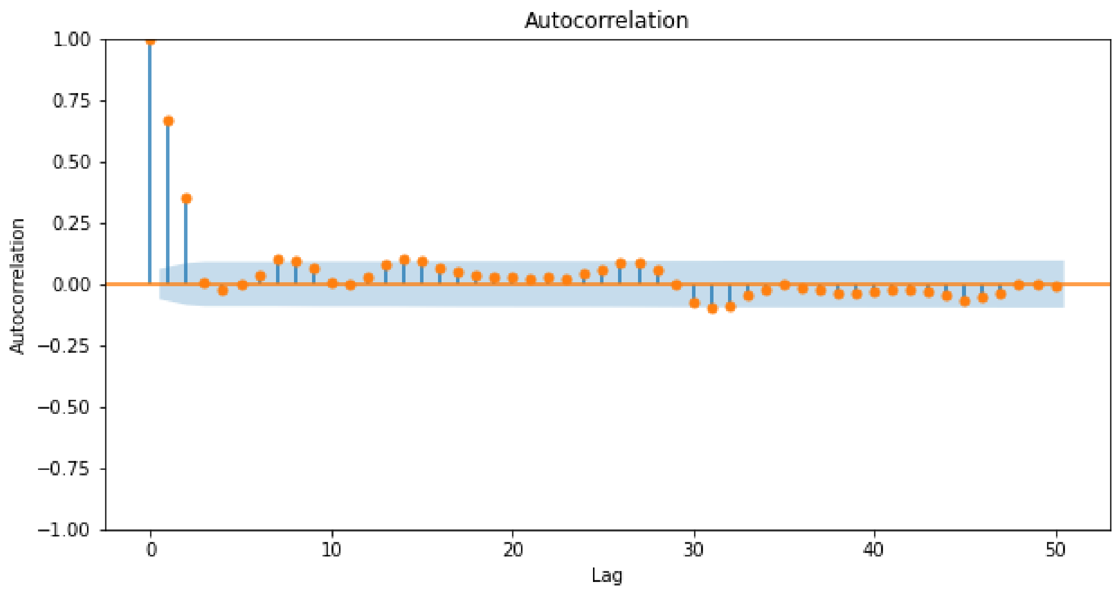


Autocorrelation RW without drift
Autocorrelation
Lag



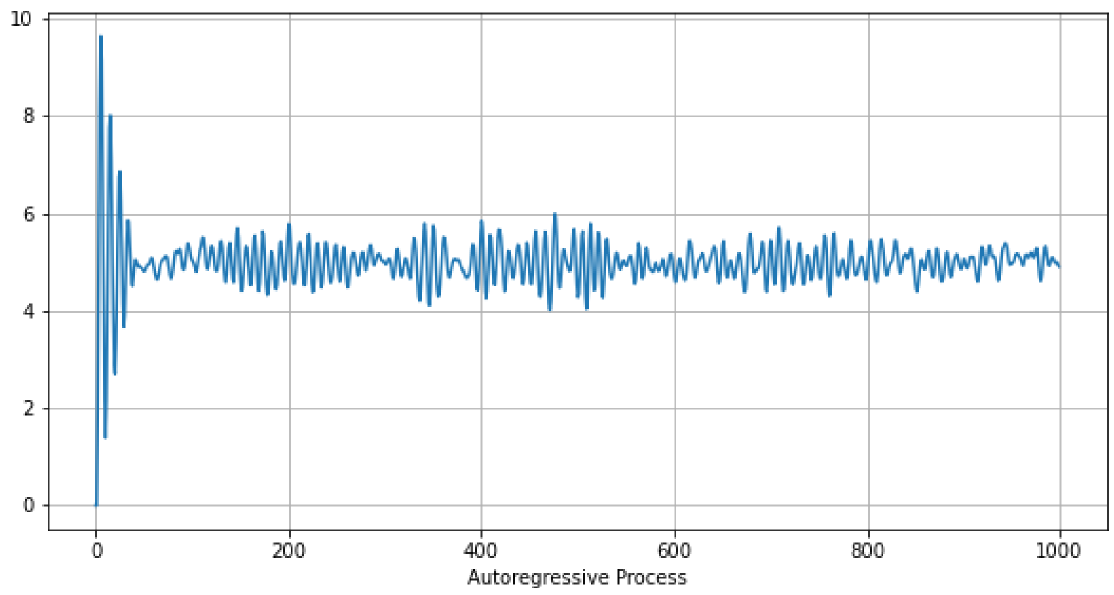
Moving Average

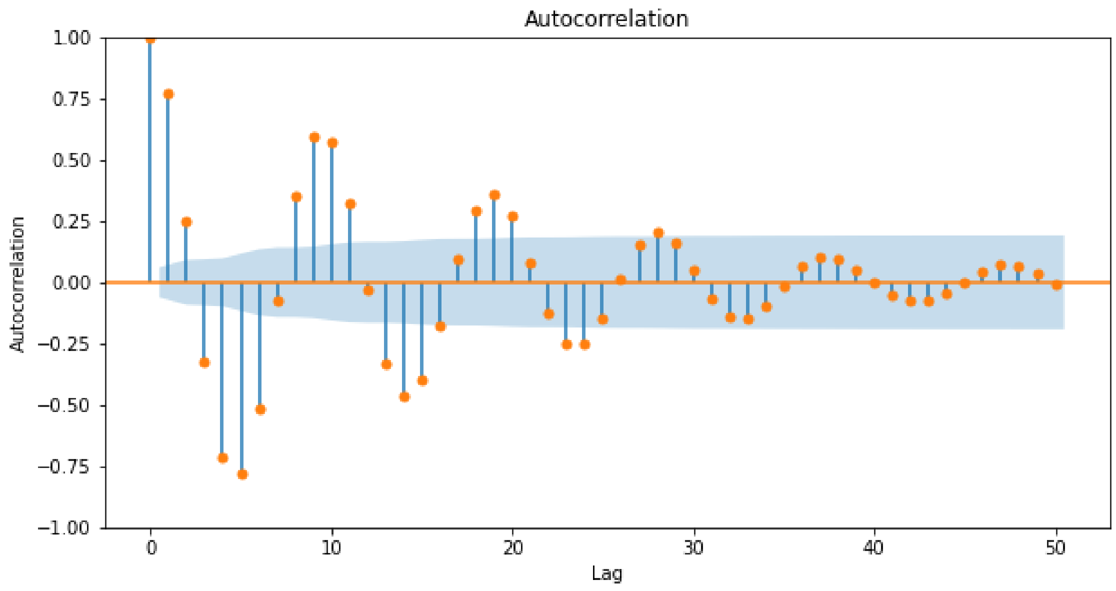
[139]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf

np.random.seed(0)

# White noise signal
w = np.random.normal(size=1000, loc=2, scale=0.1)
w = pd.DataFrame(w)
# Filter with window = 3
w = w.rolling(window=3).mean()



# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
plot_acf(w, missing='drop', lags=50, ax=ax, c='C1')
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.show()
```



Autoregressive Process

[140]:
```
# Autoregressive filter:
ar = np.zeros(1000)
for i in range(2,1000):
    ar[i] = 1.5 * ar[i-1] - 0.9 * ar[i-2] + w.iloc[i]

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(ar)
ax.grid()
plt.xlabel("Autoregressive Process")

plt.show()

fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
plot_acf(ar, missing='drop', lags=50, ax=ax, c='C1')
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.show()
```





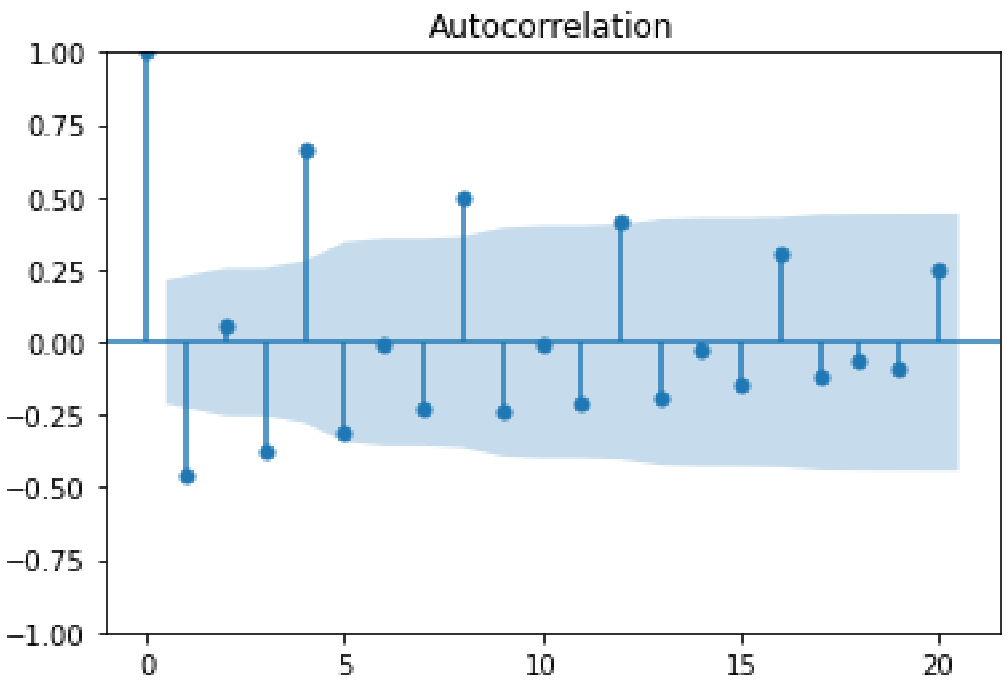
Code Example Calculating Partial Autocorrelation on log-returns

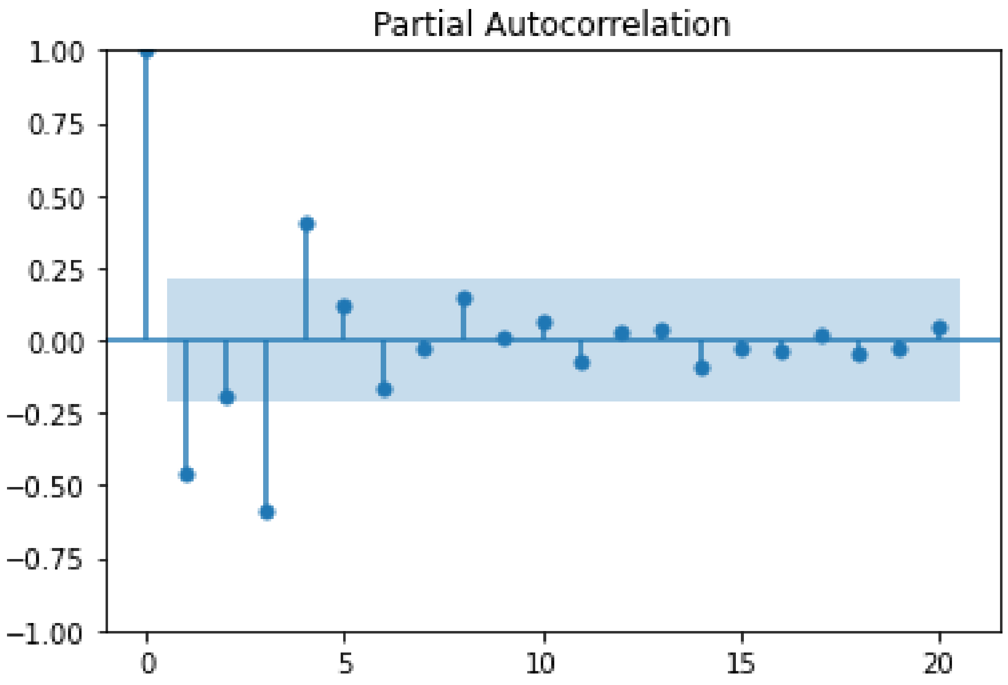
[11]:
```python
import statsmodels.api as sm
import pandas as pd
import matplotlib.pyplot as plt

# Load data
df = sm.datasets.get_rdataset("JohnsonJohnson").data
```

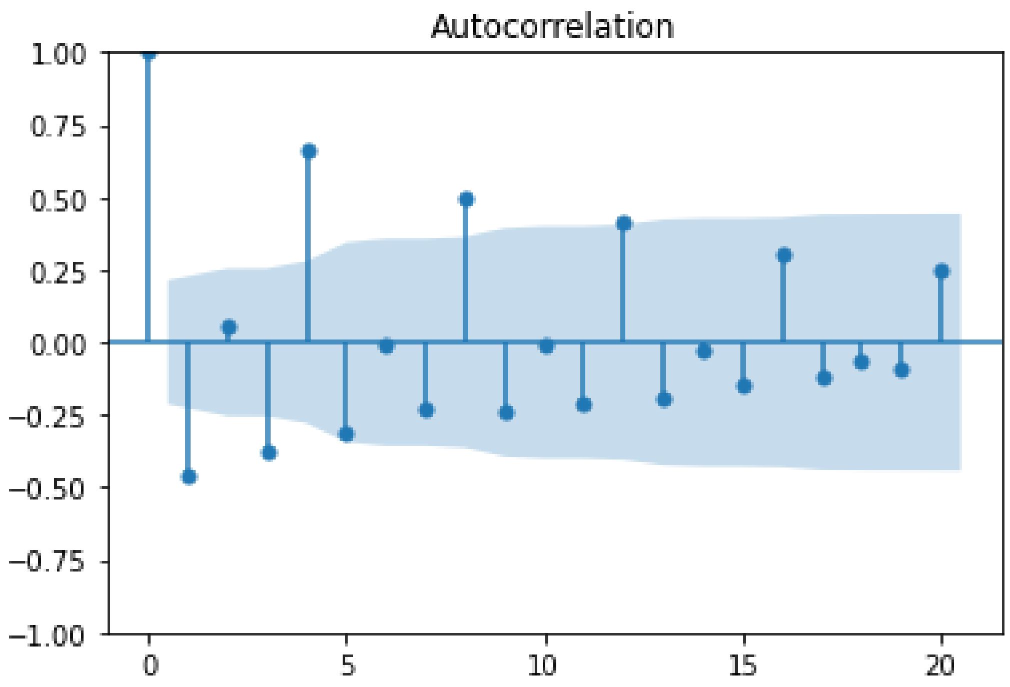
```
#The Date format on the JohnsonJohnson date is not interpretable for the index␣
  ↪function. therefore quick workaround
index = []
for i in df['time'].values:
    bla = str(i).split(".")
    if bla[1]=='0':
        index.append(str(bla[0])+'Q1')
    elif bla[1]=='25':
        index.append(str(bla[0])+'Q2')
    elif bla[1]=='5':
        index.append(str(bla[0])+'Q3')
    elif bla[1]=='75':
        index.append(str(bla[0])+'Q4')

df['time'] = index

# Create pandas DateTimeIndex
dtindexE = pd.DatetimeIndex(data=pd.to_datetime(df["time"]),
                            freq='infer')

# Set as Index
df.set_index(dtindexE, inplace=True)
df.drop("time", axis=1, inplace=True)

#log returns
logretJohnny = np.log(df["value"]) - np.log(df["value"].shift(-1))

print(f'Mean of log returns {logretJohnny.mean()}')
print(f'Variance of log returns {logretJohnny.var()}')

#tha last falue is nan, therefore it will be filled with the value before␣
  ↪(forward-fill)
logretJohnny.fillna(method='ffill')

from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.graphics.tsaplots import plot_acf

# Plot
plot_pacf(logretJohnny.fillna(method='ffill'))
plot_acf(logretJohnny.fillna(method='ffill'))
```

```
Mean of log returns -0.033666953068151505
Variance of log returns 0.04418931970538037
```

[11]:


Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20


Partial Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20



10.4 Time Series Forcasting

- Now we will model given time series data with a proper stochastic process to forecast future and unobserved values of the process
- Goal : predict future values $xn + k$ with $k = 1, 2, ...$ given a time series up to the present time $\{x_l, ..., x_n\}$

10.4.1 Procedure

There are three steps that have to be carried out subsequently, in order to achieve this goal: 1. We need to be certain that the underlying process is predictible, i.e. in the future the process will not change dramatically but continues as it has up to the present (in a probabilistic sense) 2. We choose a model class by explorative data analysis of the given time series. After that we fit the model to the training data and receive model parameters that fully describe the fitted model 3. With the fitted model we predict future values of the process

10.4.2 Autoregressive Models

The autoregressive model of order p is a discrete stochastic process that satisfies

$$X_n = a_1 X_{n-1} + a_2 X_{n-2} + \cdots + a_p X_{n-p} + W_n$$

where $a_1, a_2, \dots, a_n$ are the model parameters and $W_1, W_2, \dots$ is a white noise process with variance σ^2. For a given autoregressive process, the characteristic polynomial is defined as

$$\Phi(x) = 1 - a_1 x - a_2 x^2 - \cdots - a_p x^p.$$

Autoregressive models are **exclusively** used for modeling of **stationary** processes

Example: AR(1) $\to$ Autoregressive process of order 1

- defined by:
$$X_n = a_1 X_{n-1} + W_n$$
- special case of an AR(1) with a_1 is a random walk but non-stationary
- Expected value of the process by taking expectations on both sides of the equation: $X_n = a_1 X_{n-1} + W_n$
$$\mu = \mathrm{E}\left(X_n\right) = a_1 \mathrm{E}\left(X_{n-1}\right) + \mathrm{E}\left(W_n\right) = a_1 \mu + 0$$
- Thus, $\left(a_1 - 1\right)\mu = 0$ which implies that $\mu = 0$, if $a_1 \neq 1$
- Conclusion :
  - I.e. **if** the process is **stationary**, then the mean function is $\mu(i) = 0$
  - **If** the process is a **random walk**, i.e. non-stationary, with $a_1 = 1$, then μ may take on any value
- Next we compute the variance of the process:
$$\sigma_X^2 = \operatorname{Var}\left(X_n\right) = a_1^2 \operatorname{Var}\left(X_{n-1}\right) + \operatorname{Var}\left(W_n\right) = a_1^2 \sigma_X^2 + \sigma^2$$
- So we find that $\sigma_X^2 = a_1^2 \sigma_X^2 + \sigma^2$ or, after some rearrangements:
$$\sigma_X^2 = \frac{\sigma^2}{1 - a_1^2}$$
  - a_1^2 cannot be 1 or larger than 1. Larger 1 means negative variance, which is bad
- To have a non-negative and constant value for the variance σ_X^2, the absolute value $|a_1|$ must be less than 1
- Interpretation : in order to be stationary, the dependence of the process on past values should not be too strong.

General Condition for an AR(p) to be Weakly Stationary An $AR(p)$ stochastic process is weakly stationary, if all (complex) roots of the characteristic polynomial
$$\Phi(x) = 1 - a_1 x - a_2 x^2 - \cdots - a_p x^p$$
exceed 1 in absolute value.

Example AR(1)

- characteristic polynomial is given by
$$\Phi(x) = 1 - a_1 x$$
- Single root of the polynomial Φ is
$$x = 1/a_1$$
which exceeds 1 in absolute value if and only if $|a_1| < 1$

Example: AR(3) Process

- $AR(3)$ process
$$X_n = 0.5X_{n-1} - 0.5X_{n-2} - 0.1X_{n-3} + W_n$$
- Roots of characteristic polynomial : $1.28, 1.28, 6.09 \to$ stationary
- Simulation of AR(3) process:

Code Example Characteristic roots We study the AR(3) process

$$X_n = 0.5X_{n-1} - 0.5X_{n-2} - 0.1X_{n-3} + W_n$$

We compute the absolute values of the roots of the characteristic polynomial

$$\Phi(x) = 1 - 0.5x + 0.5x^2 + 0.1x^3$$

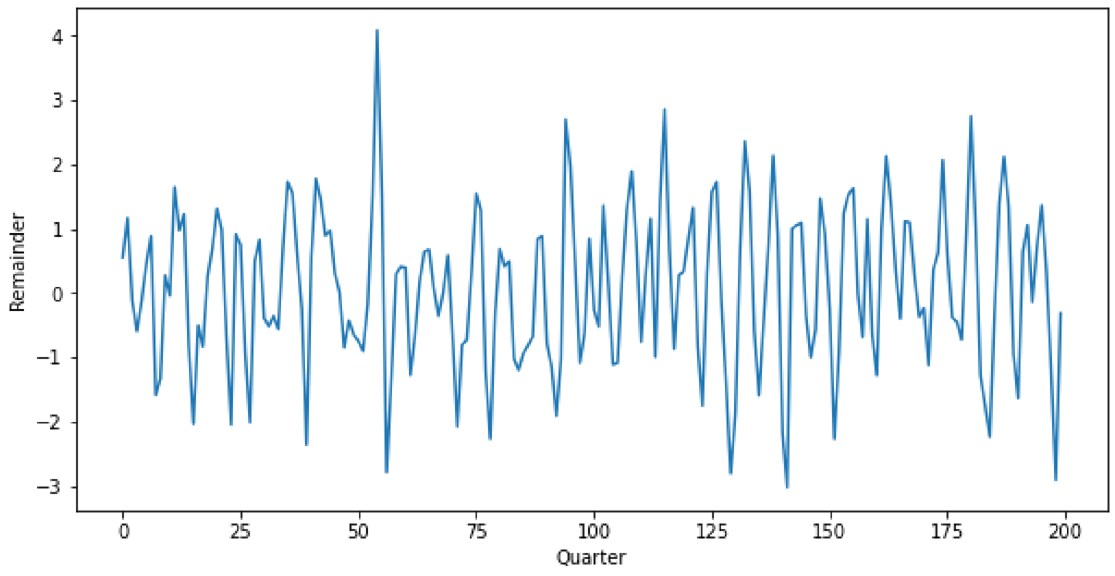
with the following command

[141]:
```
import numpy as np

# note order: p[0] * x**n + p[1] * x**(n-1) + ... + p[n-1]*x + p[n]
abs_roots = abs(np.roots([0.1, 0.5, -0.5, 1]))
print(abs_roots)
```

```
[6.09052833 1.28136399 1.28136399]
```

We can see that all the values are larger than 1 and hence the process is stationary. We can simulate a time series from this model with the `arima_process.ArmaProcess()` method from `statsmodels.tsa`.

[142]:
```
import matplotlib.pyplot as plt
from statsmodels.tsa.arima_process import ArmaProcess

# Simulate time series using ArmaProcess
ar3 = [1, -0.5, 0.5, 0.1]
simulated_data = ArmaProcess(ar3, ma=[1])
simulated_data = simulated_data.generate_sample(nsample=200)

# plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(simulated_data)
plt.xlabel("Quarter")
plt.ylabel("Remainder")

plt.show()
```



The figure shows an AR(3) realization of the code above.

Autocorrelation of AR(p) processes

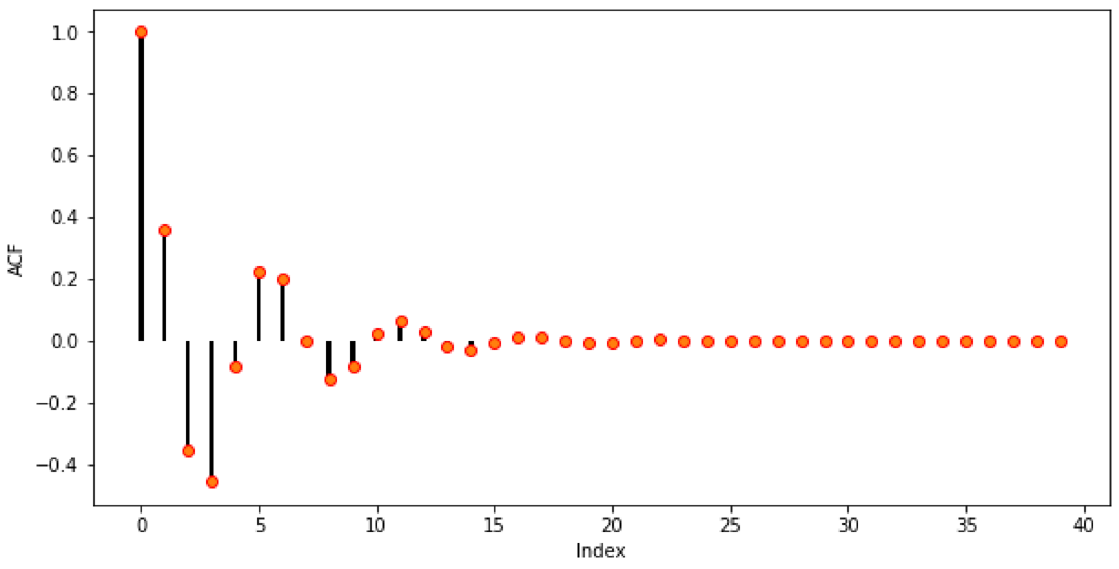
- Questions before fitting an autoregressive model
  1. Is the autoregressive model the right choice for the given data?
  2. what is the proper model order p for the given data?
- **sample autocorrelation** is compared with the theoretical autocorrelation of the AR(p) model.
  - with that, we can judge if the time series is autoregressive
- **partial autocorrelation** for determining the order of an autoregressive model

Code Example Theoretical autocorrelation of AR(p) Process

- AR(1) process: theoretical autocorrelation at lag h is a_1^h
- autocorrelation decays exponentially fast
- AR(3) Process:
- The process is defined by the coefficients $a_1 = 0.5$, $a_2 = -0.5$ and $a_3 = -0.1$. We use `ArmaProcess()` to simulate the data, and `.acf()` to find the autocorrelation function.

[143]:
```
import matplotlib.pyplot as plt
from statsmodels.tsa.arima_process import ArmaProcess
import numpy as np
# Compute the theoretical autocorrelation function
ar3 = [1, -0.5, 0.5, 0.1]
lag = 40
acf_theor = ArmaProcess(ar = ar3)
acf_theor = acf_theor.acf(lag)
```

```python
# Plot
x = np.arange(lag)
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
plt.bar(x, acf_theor, width=0.2, color="black")
plt.plot(x, acf_theor, "ro", markerfacecolor="C1")
plt.xlabel("Index")
plt.ylabel("ACF")

plt.show()
```



- As it can be seen the autocorrelation of the given AR(3) is **oscillating** and **decreasing** essentially following an exponential function.
- This is the typical autocorrelation behaviour of an autoregressive process.
- therefore the AR as a model might be a good choice

Partial autocorrelation

- Previous examples indicate that the autocorrelation of an $AR(p)$ process is nonzero for a wide span of lags
- This is due to propagation of correlation through the model: If X_k is strongly correlated with X_{k+1} and in turn X_{k+1} is strongly correlated with X_{k+2}, then X_k will likely be correlated with X_{k+2} as well
- If we want to study the direct correlation between X_k and X_{k+2}, i.e. the proportion of correlation that is not due to X_{k+1}, we have to compute the partial autocorrelation
- For a weakly stationary stochastic process $\{X_1, X_2, ...\}$ the partial autocorrelation is defined as

$$\pi(h) = \text{Cor}\left(X_k, X_{k+h} \mid X_{k+1}, \dots, X_{k+h-1}\right)^a$$

[a] The quantity $\mathrm{Cor}(X, Y \mid Z)$ denotes the conditional correlation of X and Y given the value of Z

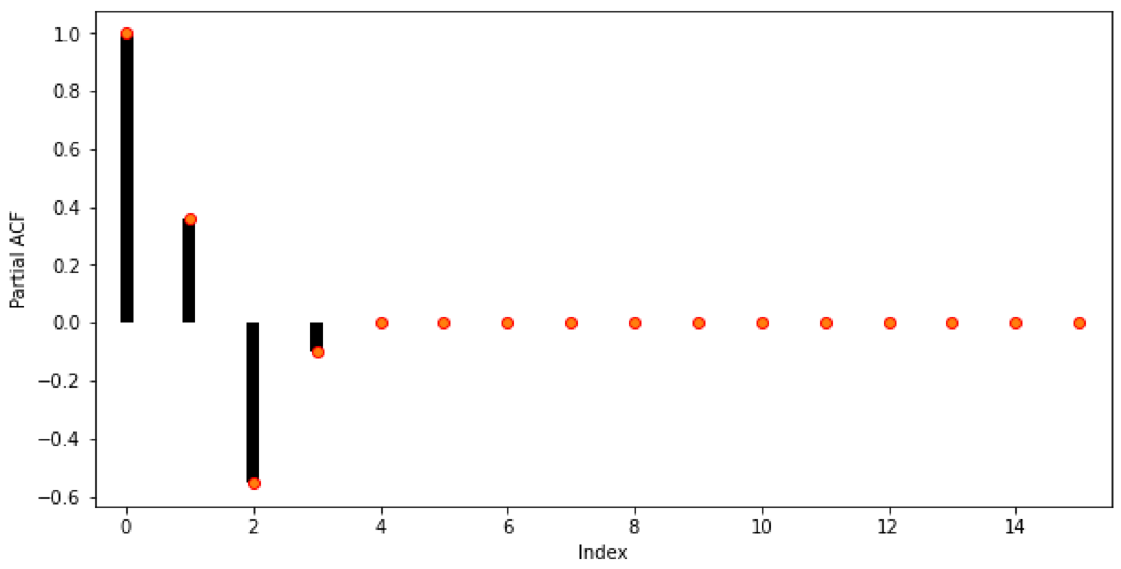
- two important properties
  1. the i-th coefficient α_i of an AR(p) process equals π_p, i.e. the partial autocorrelation value at lag p of the process
  2. for an AR(p) the partial autocorrelation at lags greater than p is zero, i.e. $\pi_k = 0$ for $k > p$

Order of an AR(p) Process

- How do we investigate a given autoregressive time series and determine the order of the model?
  1. Compute the partial autocorrelation
  2. Choose the largest lag k for which the value $\pi(k)$ is not zero $\rightarrow$ order p of AR(p) model

Code Example Order of an AR(p) process We consider the time series generated from an AR(3) process.

[144]:
```
# Compute the partial autocorrelation function
pacf_theor = ArmaProcess(ar=ar3)
pacf_theor = pacf_theor.pacf(lag)

# Plot
x = np.arange(lag)
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.bar(x, pacf_theor, width=0.2, color="black")
ax.plot(x, pacf_theor, "ro", markerfacecolor="C1")
ax.set_xlim([-0.5, 15.5])
plt.xlabel("Index")
plt.ylabel("Partial ACF")
plt.show()
```

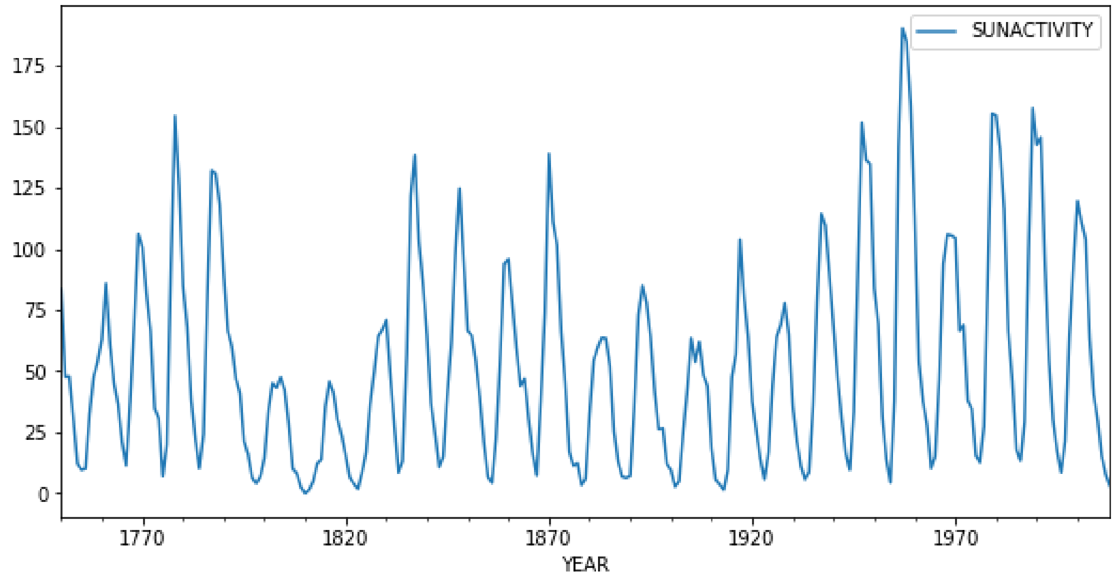


As it can be seen, the partial autocorrelation coefficients larger than 3 are almost zero. So in practice, i.e. when only the time series is at hand, we would choose an autoregressive model of order 3 for modelling the present sequence.

Code Example Determining an AR(p) process Forecasting solar activity is important for satellite drag, telecommunication outages and solar winds in connection with blackout of power plants. As an indicator of solar activity the sunspot number is used, among others. The Swiss astronomer Johann Rudolph Wolf introduced the sunspot number in 1848 and the number of sunspots is now known on a monthly basis back to the year 1749 (from 1749 to 1848 the data is questionable). The data set is built-in in **Python** and the following code produces the plot:

[145]:
```
import statsmodels.api as sm
import pandas as pd
import matplotlib.pyplot as plt

# Load data
dta = sm.datasets.sunspots.load_pandas().data

# Create pandas DateTimeIndex
dtindex = pd.DatetimeIndex(data=pd.to_datetime(dta["YEAR"], format="%Y"),
                            freq='infer')
# Set as Index
dta.set_index(dtindex, inplace=True)
dta.drop("YEAR", axis=1, inplace=True)
# Select only data after 1750
dta = dta["1750":"2008"]

# Plot
```

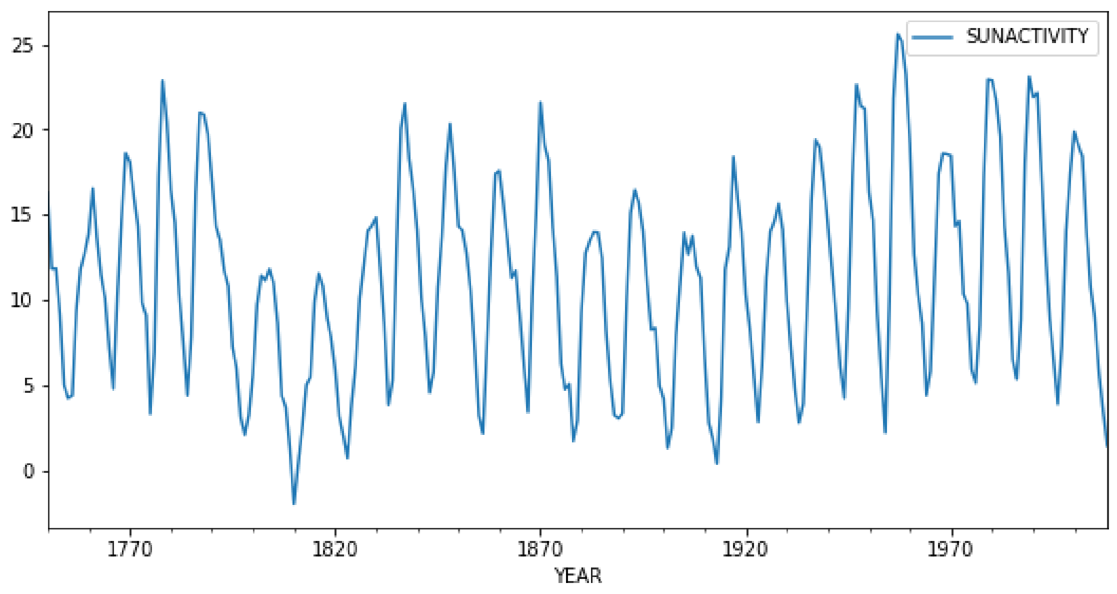
```
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
dta.plot(ax=ax)
plt.show()
```



It is important to note that the sunspot data is *not periodic*, i.e. the cycle duration is not constant. So the quasi periodic behaviour must not be mistaken for seasonality. The peaks and minima of the series are not known in advance.

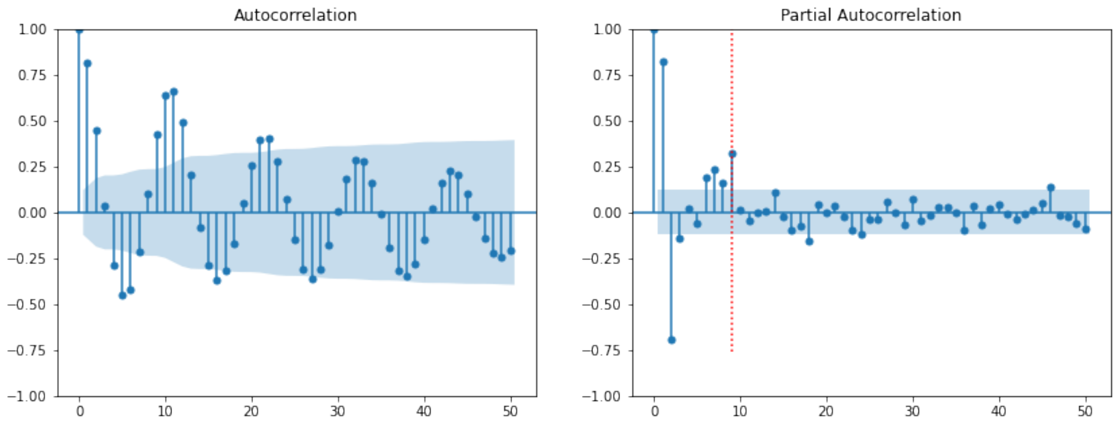
Eyeballing indicates that the underlying process is not stationary: There are clearly phases with different variances and means. We perform a Box-Cox square-root transform to stabilize the variance.

[146]:
```
import numpy as np

# Box-Cox square root transformation
dta_sq = (np.sqrt(dta) - 1) * 2

# Plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
dta_sq.plot(ax=ax)

plt.show()
```



The square-root transformed sequence is shown. The variance is visually stabilized and although there is still some trend visible, the series looks fairly stationary after the transformation. Next, we compute the autocorrelation and partial autocorrelation functions in order to clarify whether the autoregressive model is the right choice and in case to determine the proper model order.

[147]:
```
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.graphics.tsaplots import plot_acf

# Plot
fig = plt.figure(figsize=(14, 5))
ax1 = fig.add_subplot(1, 2, 1)
plot_acf(dta_sq, lags=50, ax=ax1)
ax2 = fig.add_subplot(1, 2, 2)
plot_pacf(dta_sq, lags=50, ax=ax2)
ax2.plot([9, 9], [-0.76, 1], ':r')

plt.show()
```



The autocorrelation (left) and partial autocorrelation (right) are depicted. As it can be seen, the autocorrelation shows the typical behaviour of an autoregressive process: an oscillating pattern with an exponential decay. The partial autocorrelation shows that the direct dependency has a maximum lag of 9 which we will use as our model parameter p in the next section. (Model Fitting procedure) The red dotted line marks this threshold.

10.4.3 Model Fitting

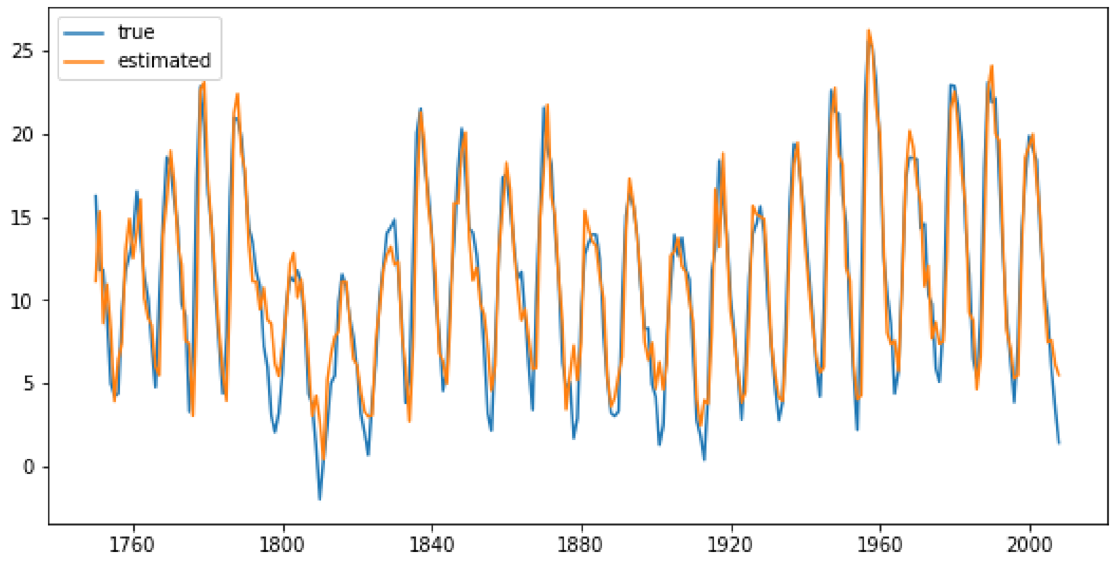
- Task: estimation of an AR(p) model from given time series data
- Assumption: Time series $\{x_1, ..., x_n\}$ is a realization of a **stationary** process
- Fitting procedure
  1. Is the time deries an autoregressice process
     - Investigate autocorrelation and partial autocorrelation
     - is the autocorrelation decaying exponentially towards 0 and possibly oscillates?
     - is the partial autocorrelation zero for larger legs?
     - autoregressive model is appropriate
  2. Choose the largest lag p, such that the partial autocorrelation is not zero
  3. choose parameters $a_1, ..., a_p$ such that given data approximates the realization of the autoregressive process
- Given the data $\{x_1, x_2, \dots, x_n\}$ and a model order p, we fit the AR(p) process to the data by solving the following linear equation system

$$\begin{aligned}x_{p+1} &= a_1 x_p + a_2 x_{p-1} + \dots + a_p x_1 + W_{p+1}\\ x_{p+2} &= a_1 x_{p+1} + a_2 x_p + \dots + a_p x_2 + W_{p+2}\\ &\vdots\\ x_n &= a_1 x_{n-1} + a_2 x_{n-2} + \dots + a_p x_{n-p} + W_n\end{aligned}$$

- System is solved in the least squares sense with the solution : estimates $\hat{a}_1, \dots, \hat{a}_p$
- 3 further standard methods for estimating the $AR(p)$ coefficients:
  1. Burg's algorithm
  2. Yule-Walker
  3. Maximum Likelihood Method

Code Example Model Fitting We have a look at the sunspot number data set. We have transformed the data by a square-root and have concluded that it stems from an AR(9) model. We will now fit this model to the data

[148]:
```
from statsmodels.tsa.arima.model import ARIMA

model = ARIMA(dta_sq, order=(9, 0, 0)).fit(method="yule_walker")

# Plot
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(dta_sq, label='true')
ax.plot(dta_sq["SUNACTIVITY"] - model.resid, label='estimated')
plt.legend()
plt.show()
```



[149]:
```
print(model.summary())
```

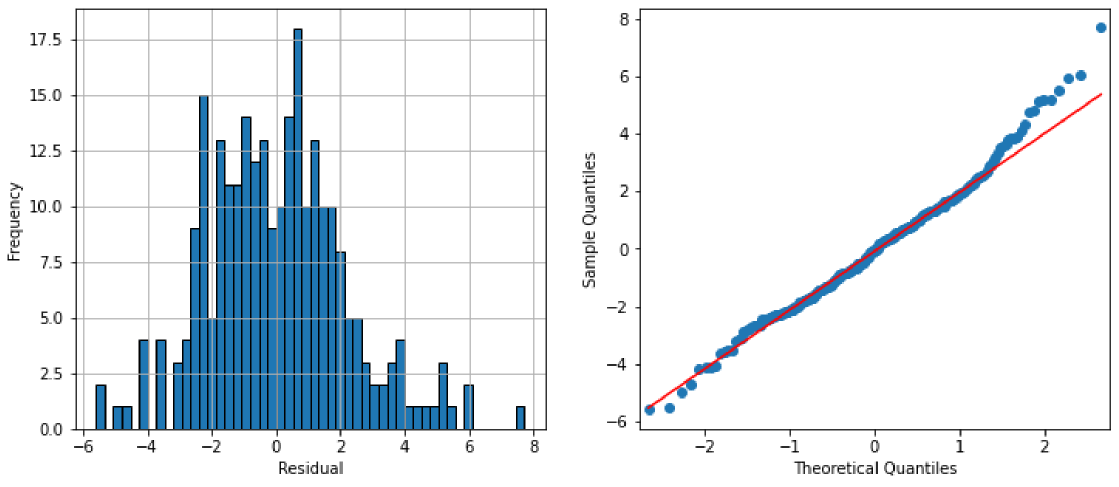
```
                               SARIMAX Results
==============================================================================
Dep. Variable:            SUNACTIVITY   No. Observations:                  259
Model:                 ARIMA(9, 0, 0)   Log Likelihood                -565.246
Date:                Tue, 28 Jun 2022   AIC                           1152.492
Time:                        10:11:36   BIC                           1191.617
Sample:                    01-01-1750   HQIC                          1168.222
                         - 01-01-2008
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
```

```
------------------------------------------------------------------------------
const         11.1370      1.211      9.200      0.000       8.764      13.510
ar.L1          1.1625      0.062     18.675      0.000       1.040       1.284
ar.L2         -0.4175      0.090     -4.658      0.000      -0.593      -0.242
ar.L3         -0.1587      0.100     -1.582      0.114      -0.355       0.038
ar.L4          0.2157      0.121      1.781      0.075      -0.022       0.453
ar.L5         -0.1946      0.124     -1.569      0.117      -0.438       0.048
ar.L6          0.0117      0.128      0.092      0.927      -0.239       0.262
ar.L7          0.1507      0.126      1.196      0.232      -0.096       0.398
ar.L8         -0.2071      0.111     -1.872      0.061      -0.424       0.010
ar.L9          0.2989      0.066      4.548      0.000       0.170       0.428
sigma2         5.0341      0.473     10.645      0.000       4.107       5.961
================================================================================
===
Ljung-Box (L1) (Q):                   0.59   Jarque-Bera (JB):
9.46
Prob(Q):                              0.44   Prob(JB):
0.01
Heteroskedasticity (H):               0.71   Skew:
0.36
Prob(H) (two-sided):                  0.12   Kurtosis:
3.59
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
```

The output above shows the 9 estimated coefficients and the estimated variance of the white noise process.

The figure shows the annually averaged time series and the output of the model. The fit seems to be reasonably accurate but we can further assess the result by examining residual plots. We choose a histogram and a qq-plot.

[150]:
```python
from statsmodels.graphics.api import qqplot

# Plot
fig = plt.figure(figsize=(12, 5))
ax1 = fig.add_subplot(1, 2, 1)
model.resid.hist(edgecolor="black", bins=50, ax=ax1)
plt.xlabel("Residual")
plt.ylabel("Frequency")
ax2 = fig.add_subplot(1, 2, 2)
qqplot(model.resid, line="q", ax=ax2)

plt.show()
```



10.4.4 Forcasting AR(p) processes

- The general methodology of forecasting stationary time series can be summarized as follows:
- k-step ahead forecast
- Assume that $\{X_1, X_2, \dots, \}$ is a stationary process and that we have observed a times series $\{x_1, x_2, \dots, x_n\}$. The k-step ahead forecast is an estimate of the random variable X_{n+k} given by
$$\widehat{X}_{n+k} = \mathrm{E}\left[X_{n+k} \mid X_1 = x_1, \dots, X_n = x_n\right].$$
Here $E[X \mid Y = y]$ denotes the conditional expectation of X given that $Y = y$
- see the example, explanation not important

Example: AR(1)

- Consider AR(1) with $|a_1| < 1$, i.e. the process is stationary:
$$X_k = a_1 X_{k-1} + W_k.$$
- If data $\{x_1, x_2, \dots, x_n\}$ is given, then we estimate the value of X_{n+1} :
$$\widehat{X}_{n+1} = \mathrm{E}\left[X_{n+1} \mid X_1 = x_1, \dots, X_n = x_n\right] = a_1 x_n.$$
- For $k > 1$ we plug-in the model equation several times and obtain
$$\begin{aligned}\widehat{X}_{n+k} &= \mathrm{E}\left[X_{n+k} \mid X_1 = x_1, \dots, X_n = x_n\right] \\ &= \mathrm{E}\left[a_1 X_{n+k-1} + W_{n+k} \mid X_1 = x_1, \dots, X_n = x_n\right] \\ &= a_1 \mathrm{E}\left[X_{n+k-1} \mid X_1 = x_1, \dots, X_n = x_n\right] \\ &= \cdots = a_1^k x_n\end{aligned}$$
  - so basically, from the last datapoint the next predictions are being done from there
- k-step ahead forecast of an AR(1) process is exponentially decaying to zero from last observation x_n and not dependent on earlier observations

10.4.5 Confidence intervals

- The standard error σ_k is the square root of the conditional variance

$$\sigma_k^2 = \operatorname{Var}\left(X_{n+k} \mid X_1 = x_1, \ldots, X_n = x_n\right)$$

- This quantity is increasing with k and converges to the process variance σ_X^2
- With this standard error a 95% confidence interval for the conditional expectation $\mathrm{E}\left[X_{n+k} \mid X_1 = x_1, \ldots, X_n = x_n\right]$ can be computed

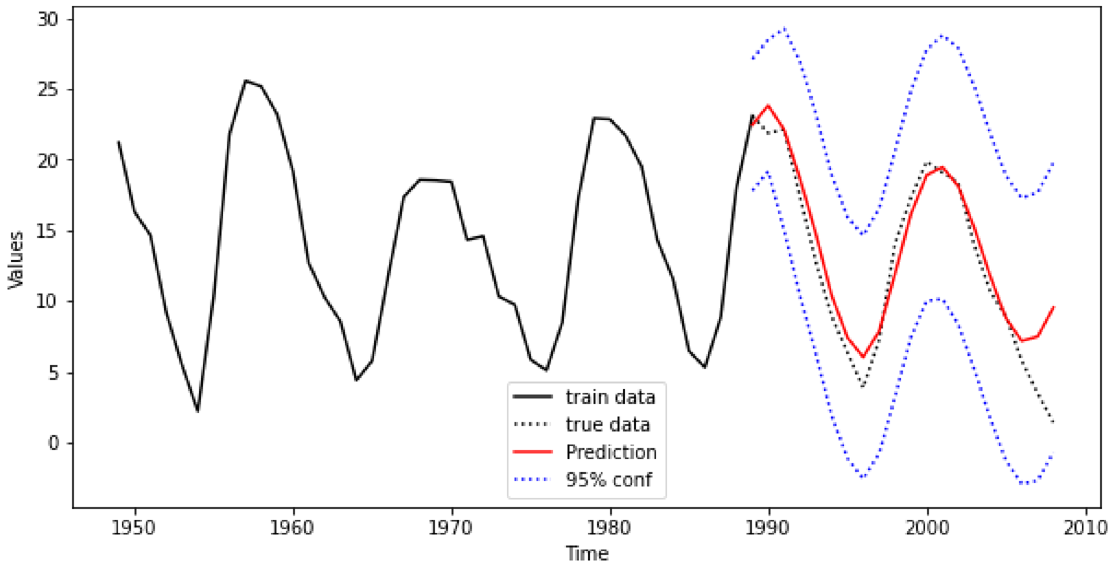
$$\widehat{X}_{n+k} \pm 1.96\sigma_k$$

10.4.6 Code Example Predictions and Confidence Intervals

We again focus on the sunspot number data. We use the annual data from 1749 until 1989 as a training set and estimate an AR(9) model from that data. We then predict the sunspot number for 25 years into the future.

[151]:
```
from statsmodels.tsa.arima.model import ARIMA
import numpy as np
import statsmodels.api as sm
import pandas as pd
import matplotlib.pyplot as plt

# Load data
dta = sm.datasets.sunspots.load_pandas().data

# Create pandas DateTimeIndex
dtindex = pd.DatetimeIndex(data=pd.to_datetime(dta["YEAR"], format="%Y"),
                            freq='infer')
# Set as Index
dta.set_index(dtindex, inplace=True)
dta.drop("YEAR", axis=1, inplace=True)
# Select only data after 1750
dta = dta["1750":"2008"]


# Box-Cox square root transformation
dta_sq = (np.sqrt(dta) - 1) * 2


# Fit model on first 130 points
model = ARIMA(dta_sq["1749": "1989"], order=(9, 0, 0)).fit(method="yule_walker")



# Predict including confidence interval
pred = model.get_prediction(start="1989", end="2008").prediction_results
pred_cov = pred._forecasts_error_cov
pred = pred._forecasts[0]
```

```
pred_upper = pred + 1.96 * np.sqrt(pred_cov[0][0])
pred_lower = pred - 1.96 * np.sqrt(pred_cov[0][0])

# Plot
x = dta_sq.index.year
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 1, 1)
ax.plot(dta_sq["1949": "1989"], '-k', label='train data')
ax.plot(dta_sq["1989": "2009"], ':k', label='true data')
ax.plot(dta_sq["1989": "2009"].index, pred, 'r', label='Prediction')
ax.plot(dta_sq["1989": "2009"].index, pred_upper, ':b', label='95% conf')
ax.plot(dta_sq["1989": "2009"].index, pred_lower, ':b')
plt.xlabel("Time")
plt.ylabel("Values")
plt.legend()
plt.show()
```



The figure shows a window of the data set from 1950 until 2014. The training set is depicted by a black solid line and ranges up to 1989. The prediction is computed and plotted for the time period 1990 to 2012 together with the true data (black dotted) that has been discarded in the modelling step. As it can be seen, there is good correspondence between true and estimated values for about 15 years. Then the prediction starts to diverge and the second minimum phase is not predicted correctly.

```
[ ]:
```