

Data Analytics with Python

Prof. Dr. Peter Roßbach

Frankfurt School of Finance & Management

Content

Preface.....	1
1 Dataframes with Pandas	3
1.1 Dataframes	3
1.2 Import and Export of Data	5
1.3 Basic Statistical Functions for Dataframes	6
2 Data Preparation	8
2.1 Handling Missing Values	8
2.2 Sampling	10
2.3 Pre-Processing Data	11
2.4 Exploratory Data Analysis	12
2.5 Visualizing Data.....	13
2.6 Transforming Categorical in Dummy Variables	17
2.7 Applying Principal Component Analysis	19
2.8 Partitioning Data.....	21
3 Classification.....	23
3.1 k-Nearest Neighbors (kNN).....	24
3.2 Naïve Bayes.....	31
3.3 Decision Trees.....	32
3.3.1 Simple Decision Trees	32
3.3.2 Random Forests	38
3.3.3 Gradient Boosted Trees	42
3.4 Discriminant Analysis.....	45
3.5 Logistic Regression.....	47
3.6 Neural Networks.....	47
3.7 Support Vector Machine.....	50
3.8 Using Cross Validation	54
3.9 Cross Validation and Imbalanced Classes.....	60
3.10 Ensemble Learning.....	63
3.11 Applying a Classification Model to New Data.....	69

4	Regression	70
4.1	Linear Regression.....	72
4.2	Ridge Regression.....	73
4.3	Support Vector Regression	75
4.4	Neural Networks.....	80
4.5	Random Forests	83
4.6	Gradient Boosting.....	85
4.7	Using Cross Validation	87
4.8	Applying a Regression Model to New Data	91
5	Interpreting Machine Learning Models	93
5.1	Skater	93
5.2	Lime.....	98
5.3	Shap	99
6	Segmentation	100
6.1	K-means	101
6.2	Hierarchical Cluster Analysis.....	105
6.3	Self-organizing Maps	107
6.4	t-SNE	112
7	Association Analysis	118

Preface

The purpose of this document is to give an introduction into the area of data analytics with Python. We will use three main libraries for data analytics provided for Python. For special cases, we will use additional libraries as mentioned in the text.

The three main libraries are

1 Numpy

NumPy is an acronym for "Numeric Python" or "Numerical Python". It is a library that provides a set of array and matrix data types which are essential for statistics, econometrics and data analysis. They behave similar to the data structures in Python. In addition, NumPy offers methods to manipulate and extract information from these structures.

An introduction into Numpy can be found in the slides "Part 2: Introduction to Python" of the course "Introduction to Programming" (first semester Bachelor at Frankfurt School).

2 Pandas

Pandas is a Python library that provides high-level data structures and a vast variety of tools for analysis. The great feature of this package is the ability to translate rather complex operations with data into one or two commands. Pandas contains many built-in methods for grouping, filtering, and combining data, as well as the time-series functionality. All of this is followed by impressive speed indicators.

In chapter 1 an introduction into using dataframes with Pandas is given.

3 Scikit-learn

Scikit-learn provides a range of supervised and unsupervised learning algorithms via a consistent interface in Python. The library is focused on modeling data. It is not focused on loading, manipulating and summarizing data. For these features, refer to NumPy and Pandas.

The structure of the implementation of the algorithms always follows the same template so that they can be used in the same way. Example for the implementation of a supervised learning algorithm:

```
import ... as ...
from ... import ...

class ModelEstimator(object):
    def __init__(self, param1 = pvalue1, ...):
        self.param1 = param1
        ...

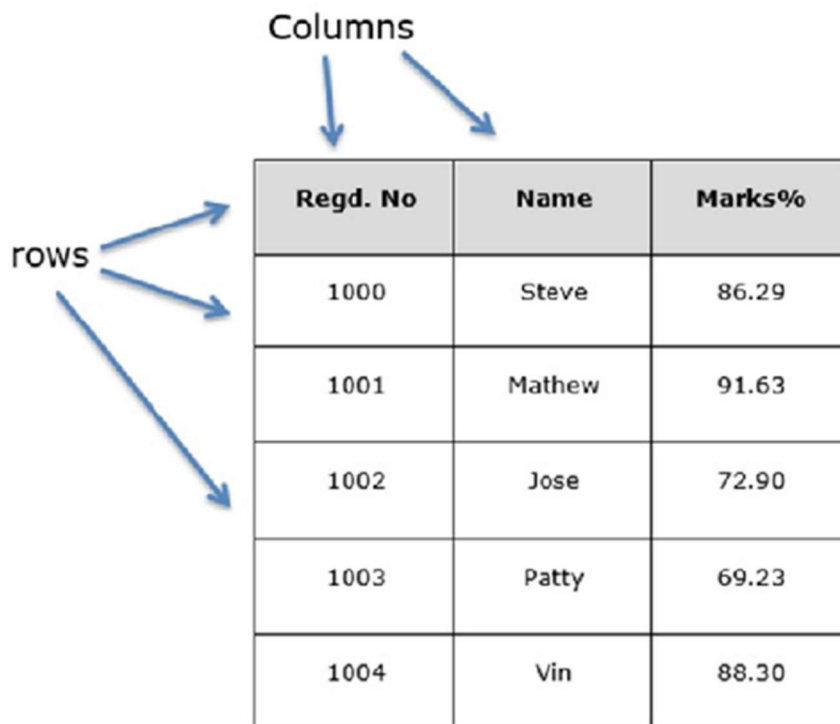
    def fit(self, X, y):
        # here the functionality for model estimation is implemented
        # X are the input values, y the target values (supervised case)
        return self
```

```
def predict(self, X):  
    # implementation of the predicting function  
    return ...
```

1 Dataframes with Pandas

1.1 Dataframes

While a matrix is of simple structure intended for mathematical operations, a dataframe is more suited to representing a table of data. It is a list of vectors of equal length. The top line of a dataframe, called the header, contains the column names. Each horizontal line afterward denotes a data row



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

For example, the following variable `df` is a dataframe containing three vectors

```
>>> import pandas as pd
>>> data = {'employee':['John Doe','Peter Gynn','Jolie Hope'],
            'salary':[21000, 23400, 26800],
            'startdate':['2010-11-1','2008-3-25','2007-3-14']}
>>> df = pd.DataFrame(data)
>>> print(df)
   employee  salary  startdate
0   John Doe   21000  2010-11-1
1  Peter Gynn   23400  2008-3-25
2   Jolie Hope   26800  2007-3-14
```

The result of this is a dataframe named `df`. The numbers on the left side of the rows are labels and **not** the indexes. You can inspect the structure using the `dtypes` property

```
>>> df.dtypes
employee      object
salary        int64
startdate     object
dtype: object
```

To retrieve data Pandas provide various variants in order to get purely integer based indexing. Like python and numpy, these are 0-based indexing. The various access variants are an integer, a list of integers, and a range of values. Examples

```
>>> df.iloc[0,1] # element in first row, second column
>>> df.iloc[:,0] # all rows, first column
>>> df.iloc[1:3,1] # second to third row, second column
>>> df.iloc[:,-2] # all rows, second column from the end
>>> df.iloc[-1,:] # last row, all columns
>>> df.iloc[[0, 2], [1, 2]] # first and third row, second to third column
```

Pandas provides too various methods of label based indexing. The loc function takes two single/list/range operators separated by a comma. The first one indicates the row and the second one indicates columns. Examples

```
>>> df.loc[1:2,:] # row with labels 1 and 2, all columns
>>> df.loc[1:2] # same as above
>>> df.loc[1:2, 'salary'] # row with labels 1 and 2, column with name
                        salary
>>> df.loc [1:2, ['employee','salary']] # second to third row,
                        columns with names employee and salary
>>> df.loc[df.loc[:,'salary']>23000] # all rows where salary>23000, all
                        columns
>>> df.loc[data['subset'] == 'train',:] # all rows where the dataframe
                        data has the value 'train' in column subset
```

The loc function can even be used to add a new row to a dataframe

```
>>> df.loc[len(df)] = ['Mike Tyson', 19400, '2018-6-27']
```

A new column can be added via

```
>>> df['age'] = [35, 42, 19, 43]
```

A column can be deleted via the drop command

```
>>> df = df.drop('age', axis = 1)
```

Two dataframes can be combined using the `concat` function. Combining two dataframes row-wise

```
>>> df1 = pd.DataFrame({'A':['AA', 'AB'], 'B':['BA', 'BB']})
>>> df2 = pd.DataFrame({'A':['CA', 'CB'], 'B':['DA', 'DB']})
>>> df3 = pd.concat([df1, df2])
>>> print(df3)
```

	A	B
0	AA	BA
1	AB	BB
0	CA	DA
1	CB	DB

Combining two dataframes column-wise

```
>>> dfa = pd.DataFrame({'A':['AA', 'AB'], 'B':['BA', 'BB']})
>>> dfb = pd.DataFrame({'C':['CA', 'CB'], 'D':['DA', 'DB']})
>>> dfc = pd.concat([dfa, dfb], axis=1)
>>> print(df3)
```

	A	B	C	D
0	AA	BA	CA	DA
1	AB	BB	CB	DB

1.2 Import and Export of Data

For most analyses, the first step involves loading a data set from a file into a dataframe. The `read_csv` function is one of the primary ways to do this. The following command will load the specified file and store it as a dataframe object called `data`

```
>>> data = pd.read_csv('path_to_file/filename.csv')
```

To avoid always specifying the path to the data file, one can set the working directory via

```
>>> import os
>>> os.chdir("D:/Path")
```

If the data is stored in an Excel file, it can be loaded via

```
>>> data = pd.read_excel('filename.xlsx')
```

The size of the dataframe can be inspected via

```
>>> print(data.shape)
(506, 12)
```

To inspect the dataframe, it is often useful to preview it with the `head` function which shows the first five rows per default or the number specified as a parameter


```
>>> print(data.head(6))
```

	Country	Area	GDP	Inflation	Life.expect	Military	\
0	Austria	83871	41600	3.5	79.91	0.80	
1	Belgium	30528	37800	3.5	79.65	1.30	
2	Bulgaria	110879	13800	4.2	73.84	2.60	
3	Croatia	56594	18000	2.3	75.99	2.39	
4	Czech Republic	78867	27100	1.9	77.38	1.15	
5	Denmark	43094	37000	2.8	78.78	1.30	

	Pop.growth	Unemployment	Type
0	0.03	4.2	train
1	0.06	7.2	test
2	-0.80	9.6	train
3	-0.09	17.7	train
4	-0.13	8.5	test
5	0.24	6.1	train

The column names can be retrieved via the *list* function

```
>>> print(list(data))
```

```
['Country', 'Area', 'GDP', 'Inflation', 'Life.expect', 'Military',
 'Pop.growth', 'Unemployment', 'Type']
```

We can use the function *to_csv* to export data

```
> data.to_csv('filename.csv')
```

1.3 Basic Statistical Functions for Dataframes

A large number of methods collectively compute base statistical measures on DataFrame. Most of them are aggregations like *sum* and *mean*. As an example for the usage of these functions, the *sum* function is used. It returns the sum of the values for the requested axis. By default, axis is the row axis (axis=0)

```
>>> print(data.sum())
```

Country	AustriaBelgiumBulgariaCroatiaCzech RepublicDen...
Area	4659831
GDP	892100
Inflation	93.6
Life.expect	2187.43
Military	44.97
Pop.growth	3.28
Unemployment	277.8
Type	traintesttraintraintesttraintesttraintesttrain...

```
dtype: object
```

The function *cumsum* has a different characteristics

```
>>> print(data.iloc[:,1:7].cumsum())
```

	Area	GDP	Inflation	Life.expect	Military	Pop.growth
0	83871.0	41600.0	3.5	79.91	0.80	0.03
1	114399.0	79400.0	7.0	159.56	2.10	0.09
2	225278.0	93200.0	11.2	233.40	4.70	-0.71
3	281872.0	111200.0	13.5	309.39	7.09	-0.80
4	360739.0	138300.0	15.4	386.77	8.24	-0.93
5	403833.0	175300.0	18.2	465.55	9.54	-0.69
6	449061.0	195700.0	23.2	539.13	11.54	-1.34
7	787206.0	231700.0	26.5	618.54	13.54	-1.27
8	1144228.0	269800.0	29.0	698.73	15.04	-1.47
9	1276185.0	296100.0	32.3	778.78	19.34	-1.41
10	1369213.0	315700.0	36.2	853.80	21.09	-1.59
11	1472213.0	353800.0	40.2	934.80	21.09	-0.92
12	1542486.0	394600.0	42.8	1015.12	21.99	0.19
13	1843826.0	425100.0	45.7	1096.98	23.79	0.57
14	1908415.0	441900.0	50.1	1169.91	24.89	-0.03
15	1973715.0	461000.0	54.2	1245.46	25.79	-0.31
16	1976301.0	541600.0	57.6	1325.21	26.69	0.83
17	2017844.0	583600.0	59.9	1406.12	28.29	1.28
18	2341646.0	637000.0	61.2	1486.44	30.19	1.61
19	2654331.0	657200.0	65.4	1562.69	32.09	1.53
20	2746421.0	680600.0	69.1	1641.39	34.39	1.71
21	2795456.0	703900.0	73.0	1717.42	35.47	1.81
22	2815729.0	732700.0	74.8	1794.90	37.17	1.62
23	3321099.0	763200.0	77.9	1876.17	38.37	2.27
24	3771394.0	803900.0	80.9	1957.35	39.87	2.44
25	3812671.0	848400.0	81.1	2038.52	40.87	3.36
26	4416221.0	855600.0	89.1	2107.26	42.27	2.73
27	4659831.0	892100.0	93.6	2187.43	44.97	3.28

The following list contains the most relevant functions and their description

Function	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of Values
median()	Median of Values
mode()	Mode of values
std()	Standard Deviation of the Values
min()	Minimum Value
max()	Maximum Value
abs()	Absolute Value
prod()	Product of Values
cumsum()	Cumulative Sum
cumprod()	Cumulative Product

2 Data Preparation

2.1 Handling Missing Values

It is quite common that some of the input records are incomplete in the sense that certain fields are missing. In a typical tabular data format, we need to validate that each record contains the same number of fields and each field contains the data type we expect. Therefore, we have the following choices: Discard the whole record if it is incomplete or fill the missing data, e.g. with the average or the median.

In Python missing values are represented by the symbol **NaN** (not a number).

Example: Reading a dataframe with missing data

```
>>> data = pd.read_csv('missingdata.csv')
>>> print(data)
```

	x	y
0	NaN	1.0
1	NaN	2.0
2	NaN	3.0
3	NaN	4.0
4	NaN	5.0
5	11.0	NaN
6	12.0	NaN
7	13.0	NaN
8	14.0	NaN
9	15.0	NaN

Checking for Missing data

```
>>> print(data.isnull())
```

	x	y
0	True	False
1	True	False
2	True	False
3	True	False
4	True	False
5	False	True
6	False	True
7	False	True
8	False	True
9	False	True

To count the NaNs column-wise in a dataframe, one can use the *sum* function in combination with the *isnull* function

```
>>> print(data.isnull().sum())
x      5
y      5
dtype: int64
```

When summing data, NaN will be treated as Zero, and when averaging, NaN is ignored

```
>>> print(data.sum())           >>> print(data.mean())
x      65                       x      13.0
y      15                       y       3.0
dtype: int64                   dtype: int64
```

If you want to simply exclude the missing values, then use the *dropna* function along with the *axis* argument. The default is *axis=0*, which means that if any value within a row is NaN then the whole row is excluded (inspect the documentation for more parameters)

```
>>> print(data.dropna())
Empty DataFrame
Columns: [x, y]
Index: []
```

To drop only the rows containing NaN in a specific column, we can use the parameter *subset*

```
>>> data.dropna(subset=['featurename'])
```

Using the parameter *thresh* removes all rows if the number of non-'NaN' values is less than the given value

```
>>> data.dropna(thresh=12)
```

To fill the missing data for example with the average, you can use the function *fillna*

```
>>> print(data.fillna(data.mean()))
      x      y
0  13.0  1.0
1  13.0  2.0
2  13.0  3.0
3  13.0  4.0
4  13.0  5.0
5  11.0  3.0
6  12.0  3.0
7  13.0  3.0
8  14.0  3.0
9  15.0  3.0
```



```
>>> print(data_ori_downsampled['Personal_Loan'].value_counts())
no      480
yes     480
Name: Personal_Loan, dtype: int64
```

An alternative variant of downsampling can be done by random selection of equally distributed line numbers from the respective categories. This procedure is particularly suitable if the target variable has more than two categories. The selection can be automated using the following function:

```
def balanced_subsample(y, size=None, random_state=None):
    # returns a List with randomly chosen row numbers
    subsample = []
    if size is None:
        n_smp = y.value_counts().min()
    else:
        n_smp = int(size / len(y.value_counts().index))
    if not random_state is None:
        np.random.seed(random_state)
    for label in y.value_counts().index:
        samples = y[y == label].index.values
        index_range = range(samples.shape[0])
        indexes = np.random.choice(index_range, size=n_smp,
                                   replace=False)
        subsample += samples[indexes].tolist()
    return subsample
```

The function is then used as follows:

```
>>> rows = balanced_subsample(data_ori['target'], random_state=0)
>>> data_downsampled = data_ori.iloc[rows,:]
```

2.3 Pre-Processing Data

For many methods in machine learning it is necessary to pre-process the data in order to transform it into a uniform range. There exist different variants for transformation. The most applied are standardization and normalization.

Standardization typically means that the range of values is standardized to measure how many standard deviations the value is from its mean. Thus, a dataframe can be standardized via

```
>>> data = (data-data.mean())/data.std()
```

Normalization typically means that the range of values are normalized to be from 0 to 1. This can be done via

```
>>> data = (data-data.min())/(data.max()-data.min())
```

Scikit-learn provides two classes for the transformation, `StandardScaler` and `MinMaxScaler`. They are implemented in the module *preprocessing*

```
>>> from sklearn import preprocessing
>>> sscaler = preprocessing.StandardScaler()
>>> data = sscaler.fit_transform(data)
>>> nscaler = preprocessing.MinMaxScaler()
>>> data = nscaler.fit_transform(data)
```

In all cases the dataframe must only contain numerical data. If there are any features containing other data type, they must be excluded from the transformation. Example:

```
>>> nscaler = preprocessing.MinMaxScaler()
>>> data.iloc[:,1:7] = nscaler.fit_transform(data.iloc[:,1:7])
>>> print(data.head(6))
```

	Country	Area	GDP	Inflation	Life.expect	Military \
0	Austria	0.135258	0.468665	0.423077	0.851372	0.186047
1	Belgium	0.046495	0.416894	0.423077	0.831555	0.302326
2	Bulgaria	0.180199	0.089918	0.512821	0.388720	0.604651
3	Croatia	0.089869	0.147139	0.269231	0.552591	0.555814
4	Czech Republic	0.126931	0.271117	0.217949	0.658537	0.267442
5	Denmark	0.067405	0.405995	0.333333	0.765244	0.302326

	Pop.growth	Unemployment	Type
0	0.427835	4.2	train
1	0.443299	7.2	test
2	0.000000	9.6	train
3	0.365979	17.7	train
4	0.345361	8.5	test
5	0.536082	6.1	train

2.4 Exploratory Data Analysis

Pandas and Scikit-learn have many useful functions to perform an Exploratory Data Analysis. The *describe* function performs a univariate non-graphical analysis for every numerical variable in a dataframe

```
>>> print(data.describe())
```

	Area	GDP	Inflation	Life.expect	Military \
count	28.000000	28.000000	28.000000	28.000000	28.000000
mean	166422.535714	31860.714286	3.342857	78.122500	1.606071
std	165538.675951	14502.115792	1.398260	3.189156	0.801352
min	2586.000000	7200.000000	0.200000	68.740000	0.000000
25%	48083.250000	20350.000000	2.575000	76.020000	1.095000
50%	87980.500000	30500.000000	3.350000	79.530000	1.500000
75%	304176.250000	38750.000000	4.025000	80.320000	1.925000

max	603550.000000	80600.000000	8.000000	81.860000	4.300000
-----	---------------	--------------	----------	-----------	----------

	Pop.growth	Unemployment
count	28.000000	28.000000
mean	0.117143	9.921429
std	0.501965	4.677872
min	-0.800000	2.800000
25%	-0.182500	6.925000
50%	0.065000	8.450000
75%	0.397500	12.725000
max	1.140000	21.700000

2.5 Visualizing Data

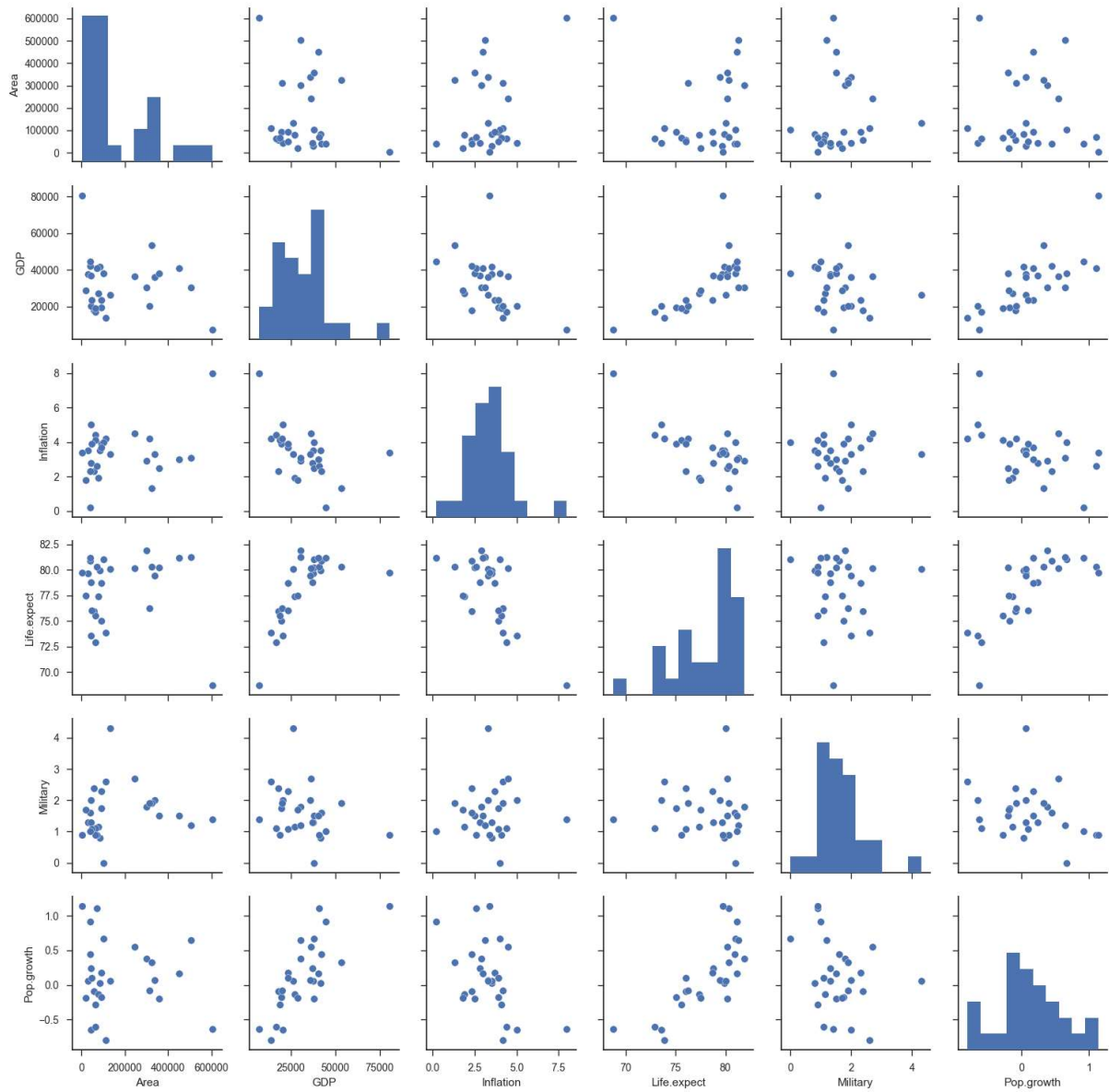
To visualize data, many libraries in Python exist. A widely used library is matplotlib which is used in this documentation too as the main library. matplotlib provides many basics for creating charts. For more sophisticated charts it is not sufficient.

Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics. Seaborn aims to make visualization a central part of exploring and understanding data. The plotting functions operate on dataframes and arrays containing a whole dataset and internally perform the necessary aggregation and statistical model-fitting to produce informative plots.

The plotting functions try to do something useful when called with a minimal set of arguments, and they expose a number of customizable options through additional parameters. Some of the functions plot directly into a matplotlib axes object, while others operate on an entire figure and produce plots with several panels. Seaborn should be thought of as a complement to matplotlib, not a replacement for it.

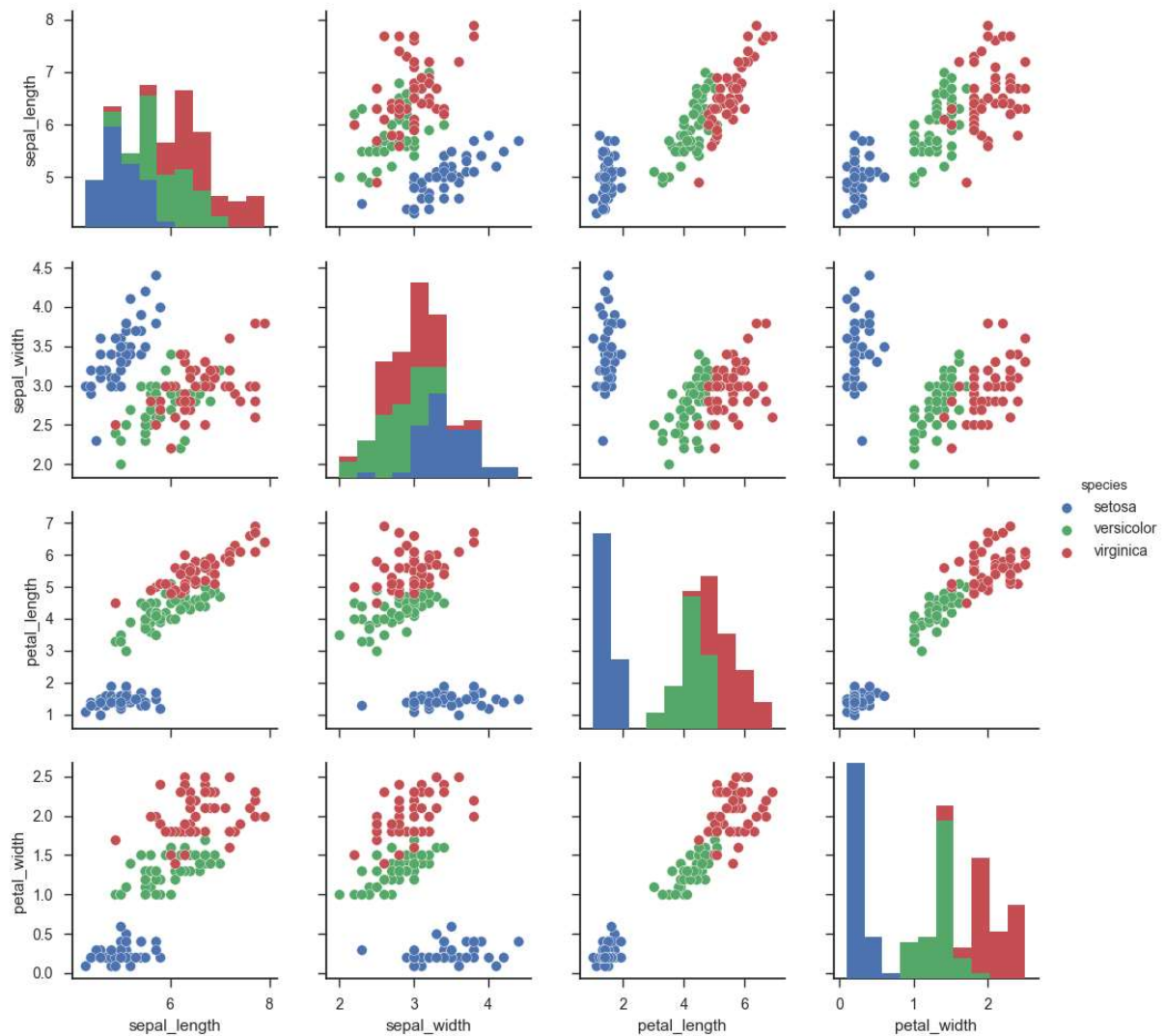
We can use the *pairplot* function to produce scatterplots of the quantitative variables

```
>>> import seaborn as sns
>>> sns.set(style="ticks")
>>> sns.pairplot(data.iloc[:,1:7])
```

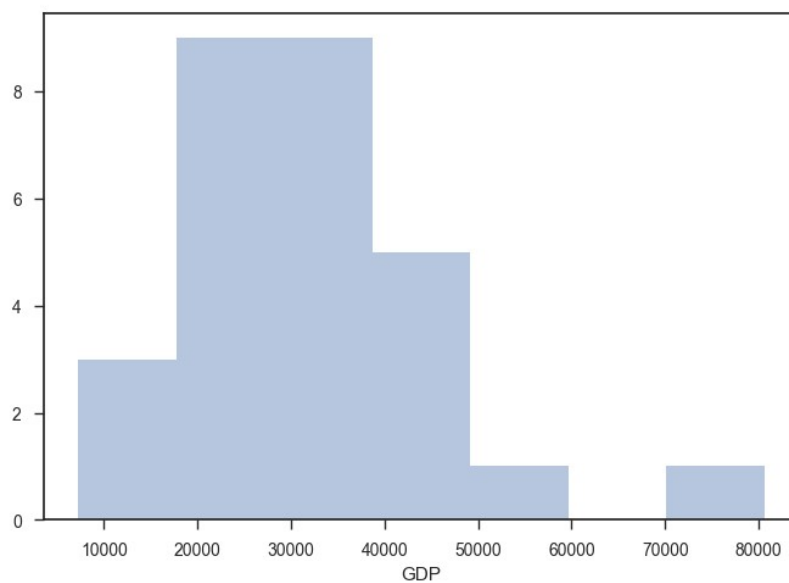
If there exists a categorical target variable, it can be used to colorize the plot. Example:

```
>>> df = sns.load_dataset("iris")
>>> sns.pairplot(df, hue="species")
```



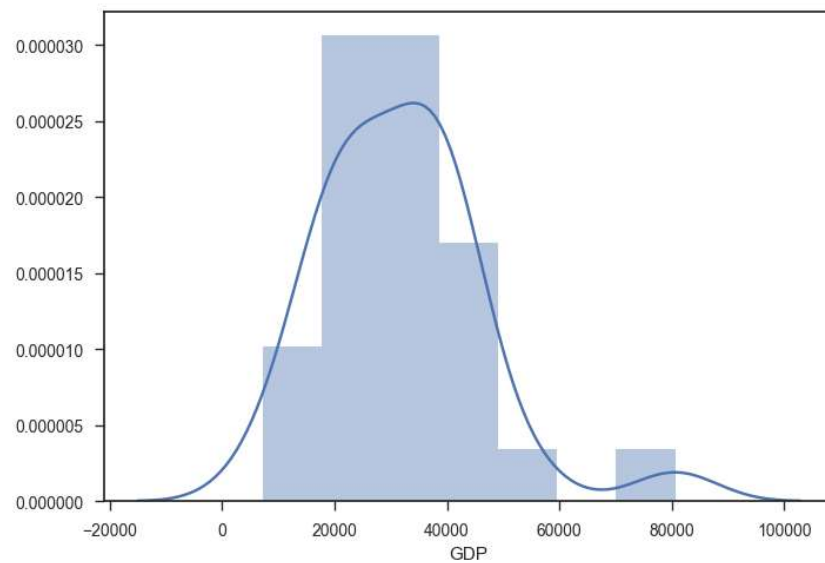
To plot a histogram, one can use the *hist* function already exists in matplotlib or the *distplot* function of seaborn

```
>>> sns.distplot(data.GDP, kde=False)
```



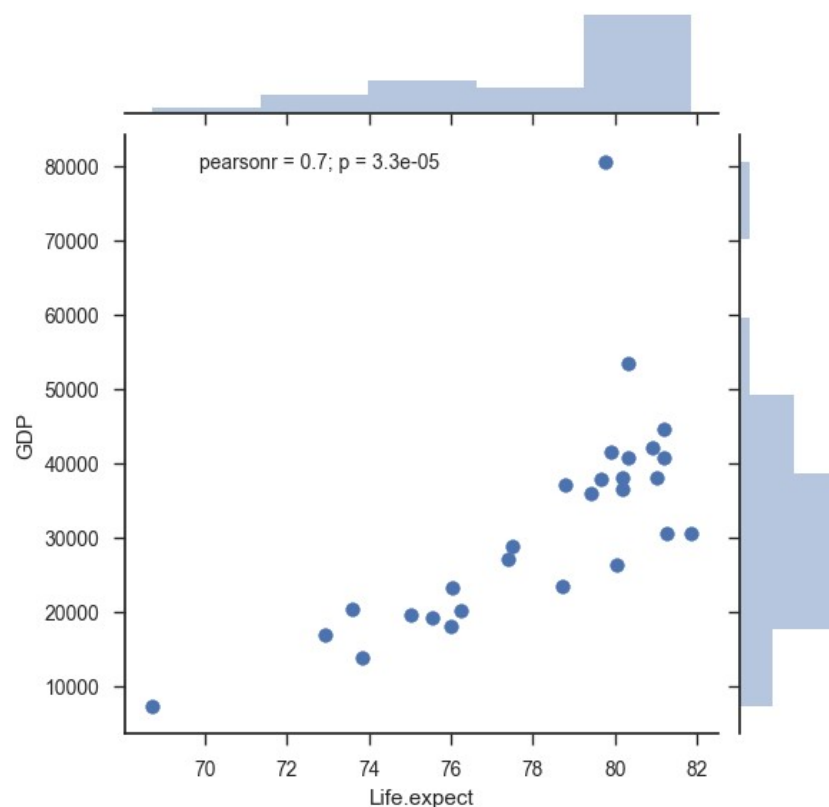
Without the parameter `kde`, the the density curve is drawn too

```
>>> sns.distplot(data.GDP)
```



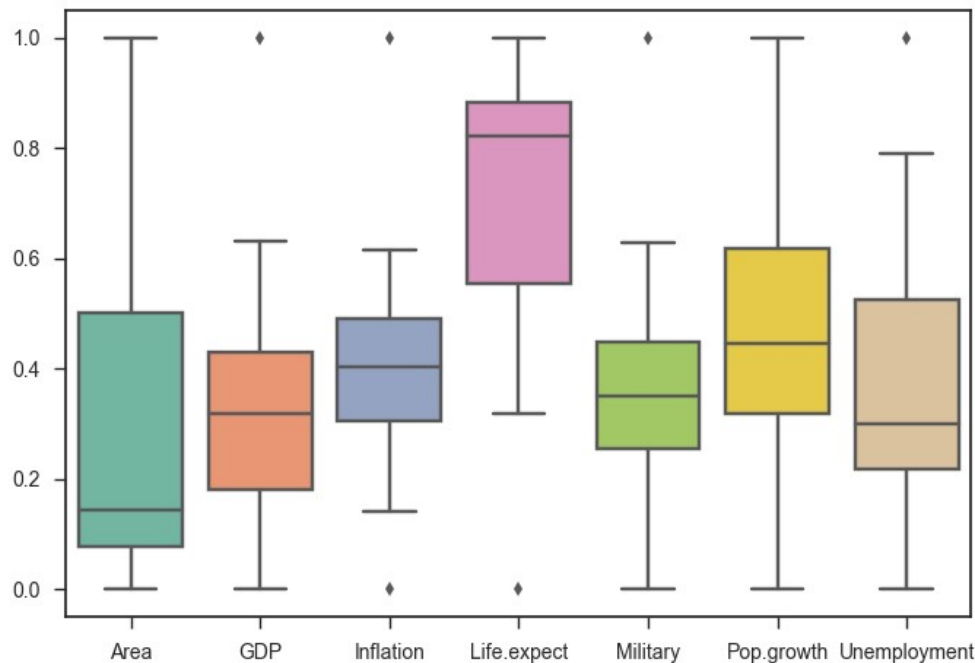
To visualize a bivariate distribution, one can use a scatterplot. It can be drawn with the matplotlib `plt.scatter` function or with the `jointplot` function from seaborn. It creates a multi-panel figure that shows both the bivariate relationship between two variables along with the univariate distribution of each on separate axes

```
>>> sns.jointplot(y="GDP", x="Life.expect", data=data)
```



To create a boxplot the *boxplot* function can be used. In this case, the usage of the normalized data makes sense to avoid effects resulting from different scale levels

```
>>> sns.boxplot(data=data_ori, orient="v", palette="Set2")
```



2.6 Transforming Categorical in Dummy Variables

Many of the methods in Machine Learning require numerical input variables. In case of categorical variables, they must be transformed in a numerical form.

For categorical variables that have two categories, e.g. female and male, the Numpy function *where()* can be used. It is identical to a classical ifelse function. The first argument is a logical expression, the second argument the value if the expression is true, and the third the value if the expression is false. In the following example, the value Female in the variable Gender will be replaced by 1 and all others by 0:

```
>>> data['Gender'] = np.where(data['Gender']=='Female', 1, 0)
```

After the replacement, the variable is automatically transformed from type object to type int32.

For categorical variables that have more than two categories, there are two approaches:

1. Label encoding: converting each value in a column to a number. Every categorical value is assigned to one numerical value, e.g. BMW -> 1, VW -> 2 etc.
2. Transforming into dummy variables: Every categorical value is assigned to an artificial binary variable. If the corresponding categorical value occurs in a data row the value of its binary replacement is equal to 1 else 0.

Scikit-learn provides classes for both methods.

We use the variable `Education` in the dataframe

```
>>> data = pd.read_csv('YoungBank_ori.csv')
>>> print(data['Education'].value_counts())
undergraduate      2096
professionell      1501
graduate           1403
Name: Education, dtype: int64
>>> print(data['Education'].head(10))
0    undergraduate
1    undergraduate
2    undergraduate
3         graduate
4         graduate
5         graduate
6         graduate
7    professionell
8         graduate
9    professionell
```

If we want to do a label encoding, we need to instantiate a *LabelEncoder* class and `fit_transform` the data:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> lb_Edu = LabelEncoder()
>>> data['Education_code'] = lb_Edu.fit_transform(data['Education'])
>>> data[['Education', 'Education_code']].head(10)
   Education  Education_code
0  undergraduate           2
1  undergraduate           2
2  undergraduate           2
3     graduate           0
4     graduate           0
5     graduate           0
6     graduate           0
7  professionell           1
8     graduate           0
9  professionell           1
```

During the transformation, the categories are first sorted alphabetically in ascending order and then the numbers are assigned.

Scikit-learn also supports binary encoding by using the *LabelBinarizer* class

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> bi_Edu = LabelBinarizer()
>>> bi_dummies = bi_Edu.fit_transform(data_ori['Education'])
>>> Edu_dummies = pd.DataFrame(bi_dummies,
                                columns=['Edu_gra', 'Edu_prof', 'Edu_under'])

>>> Edu_dummies.head(10)
```

	Edu_gra	Edu_prof	Edu_under
0	0	0	1
1	0	0	1
2	0	0	1
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0
7	0	1	0
8	1	0	0
9	0	1	0

Here too, the categories are first sorted alphabetically in ascending order and then transformed into dummy variables. To avoid the dummy variable trap, one can now drop the first column via

```
>>> Edu_dummies = Edu_dummies.drop('Edu_gra', axis = 1)
```

The resulting dataframe can now be concatenated with the original dataframe.

2.7 Applying Principal Component Analysis

To apply Principal Component Analysis we can use the PCA class of the module *sklearn.decomposition*. In the following the usage is demonstrated to a dataset, where the first ten columns contain numerical input variables and last column is a categorical target variable. It must be ignored.

First, we read and normalize the data

```
>>> data_ori = pd.read_csv('YoungBank.csv')
>>> from sklearn import preprocessing
>>> nscaler = preprocessing.MinMaxScaler()
>>> data_ori.iloc[:,0:11] = nscaler.fit_transform(data_ori.iloc[:,0:11])
>>> data = data_ori.iloc[:,0:11]
```

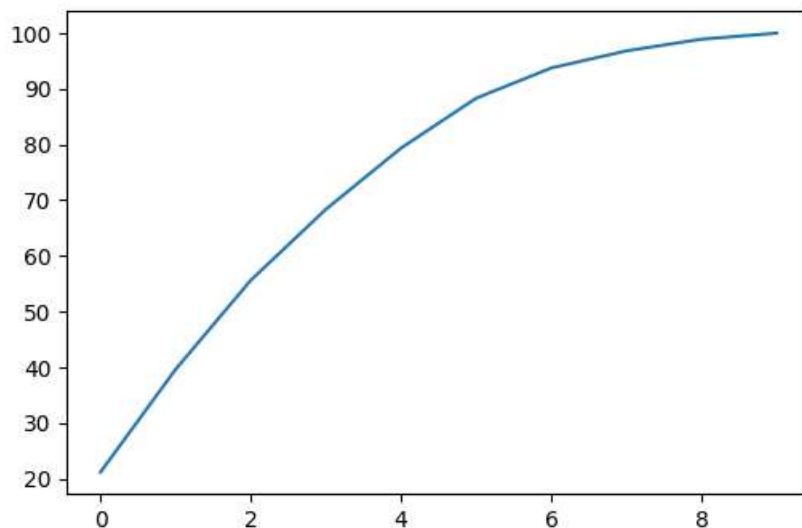
Next, we perform the Principal Component Analysis using all input variables

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=10)
>>> pca.fit(data)
```

The object variable *explained_variance_ratio_* contains the amount of variance that each principal component explains. Using the *cumsum* function, we can cumulate the variances and plot the function

```
>>> var = pca.explained_variance_ratio_
>>> cum_var = np.cumsum(np.round(pca.explained_variance_ratio_,
                                decimals=4)*100)

>>> print(cum_var)
[ 21.16  39.62  55.6   68.35  79.33  88.29  93.72  96.79  98.89  99.97]
>>> import matplotlib.pyplot as plt
>>> plt.plot(cum_var)
```



It is now the task of the data scientist to decide, how many principal components to use for the further analysis. If, for example, 88% of the total variance in the input data explained is enough, the choice would be 6 principal components. Thus, we repeat the analysis with the chosen number and transform the result in a new dataframe

```
>>> pca = PCA(n_components=6)
>>> pca.fit(data)
>>> newdata = pca.fit_transform(data)
>>> print(newdata)

[[-0.58981425 -0.16543074 -0.27658844  0.90411478  0.24461274  0.80963751]
 [-0.58429243 -0.16753443 -0.3412742   0.3812884  -0.19277169  0.81847666]
 [-0.63175829 -0.19717297 -0.54007941 -0.17010802  0.18984544 -0.086511  ]
 ...,
 [-0.65886899 -0.2186817   0.52765413 -0.47571619 -0.42253444 -0.05166712]]
```

```
[ 0.35219158 -0.35797364  0.19244649 -0.08290429 -0.63904279 -0.1359086 ]
[ 0.49583079  0.62168976 -0.32456095  0.48428779  0.35135295 -0.24289873]]
```

The component variables can now be used for the further analysis.

2.8 Partitioning Data

To partition the data into training and test set, one can use different ways. If the data already contains a label for distinguishing between train and test, the following code can be used to create the datasets

```
>>> data_ori = pd.read_csv('filename.csv')
>>> # drop subset variable and split remaining data into X and Y
>>> X = data_ori.drop('subset', axis = 1)
>>> X = X.drop('target', axis = 1)
>>> Y = data_ori['target']
>>> # partition into train and test set
>>> X_train = X.loc[data_ori['subset'] == 'train',:]
>>> Y_train = Y.loc[data_ori['subset'] == 'train']
>>> X_test = X.loc[data_ori['subset'] == 'test',:]
>>> Y_test = Y.loc[data_ori['subset'] == 'test']
```

In this case the labels are stored in a variable *subset* and the Y-variable is named *target*.

Another way is to split according to a specific criterion. In the following case all rows with an even number are used for the training and the others for the test set

```
>>> evens = [n for n in range(X.shape[0]) if n % 2 == 0]
>>> X_train = X.iloc[evens,:]
>>> Y_train = Y.iloc[evens]
>>> odds = [n for n in range(X.shape[0]) if n % 2 != 0]
>>> X_test = X.iloc[odds,:]
>>> Y_test = Y.iloc[odds]
```

Scikit-learn offers a function *train_test_split* to partition on base of a random sampling. The function automatically returns all 4 datasets

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                                         test_size=0.25, random_state=0)
```

In the above case, the size of the test set is 25% of the original size. Thus, 75% are used for the training set. The parameter *random_state=0* guaranties that the same random numbers are created every run with a seed of 0.

To create a stratified partition process with the same class distribution in every dataset, one can use the `stratify` parameter. The value is the name of the class variable

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                                         stratify=Y, test_size=0.25, random_state=0)
```

3 Classification

Scikit-learn provides various methods of classification. They are organized in different modules and classes.

In this chapter, we use the case of a telecommunication company that want to build a strategy for customer retention to preserve customer lifetime value. A first important step in such a strategy is to identify those customers that are likely to terminate their contract (churn) which is a typical classification task.

The data is stored in a CSV file and contains the following features:

Name	Description	Measurement	Type
CHURN	Churning	True/False	Boolean
ACLENGTH	Account length	Days account has been active	Metric
INTPLAN	International plan	INTPLAN=1 for YES	Dummy
VMPLAN	Voice mail plan	VMPLAN=1 for YES	Dummy
NVMAIL	Number of vmail messages	Count	Metric
DMIN	Total day minutes	Count	Metric
DCALL	Total day calls	Count	Metric
DCHARGE	Total day charge	US dollars	Metric
EMIN	Total evening minutes	Mins	Metric
ECALL	Total evening calls	Count	Metric
ECHARGE	Total evening charge	Count	Metric
NMIN	Total night minutes	Mins	Metric
NCALL	Total night calls	Count	Metric
NCHARGE	Total night charge	Count	Metric
IMIN	Total international minutes	Mins	Metric
ICALL	Total international calls	Count	Metric
ICHARGE	Total international charge	Count	Metric
CUSCALL	Calls to customer service	Count	Metric

We begin with importing the libraries and modules that are generally needed in the classification application

```
>>> import os
>>> import numpy as np
>>> import pandas as pd
>>> import sklearn as sk
>>> import matplotlib.pyplot as plt
>>> from sklearn.metrics import accuracy_score, confusion_matrix
```

As a next step, the path to the directory of the file is set and the file is read into a Pandas dataframe

```
>>> os.chdir("D:/Path")
>>> data_ori = pd.read_csv('churn_balanced.csv')
```

Next, we split the data into inputs X and the target variable Y

```
>>> X_ori = data_ori.drop('CHURN', axis = 1)
```

```
>>> Y = data_ori['CHURN']
```

Now, we normalize the input data

```
>>> X = (X_ori-X_ori.min())/(X_ori.max()-X_ori.min())
```

For the creation of the classification models we need to partition the data into a training set and a test set. In this case, we use a simple algorithm. All the rows having an uneven number will be assigned to the training set and all the others to the test set

```
>>> evens = [n for n in range(X_ori.shape[0]) if n % 2 == 0]
```

```
>>> X_train = X.iloc[evens,:]
```

```
>>> Y_train = Y.iloc[evens]
```

```
>>> print(Y_train.value_counts())
```

```
no      258
```

```
yes     242
```

```
Name: CHURN, dtype: int64
```

```
>>> odds = [n for n in range(X_ori.shape[0]) if n % 2 != 0]
```

```
>>> X_test = X.iloc[odds,:]
```

```
>>> Y_test = Y.iloc[odds]
```

```
>>> print(Y_train.value_counts())
```

```
no      258
```

```
yes     242
```

```
Name: CHURN, dtype: int64
```

For classification the basic procedure is the same for all different approaches. At first, the model is created using the training data. Then, the model is applied using the test data to evaluate its performance. When indicated, the model is re-trained tuning some parameter of the algorithm to increase the quality.

In the following, we will apply the methods in a competitive manner. To compare the results, we create a dataframe where we store the method and the accuracies of the training and the test sets as a measure of classification quality. The dataframe is created via

```
>>> report = pd.DataFrame(columns=['Model', 'Acc.Train', 'Acc.Test'])
```

3.1 k-Nearest Neighbors (kNN)

To create a kNN classifier you can apply the module *sklearn.neighbors.KNeighborsClassifier*, which uses the Euclidian distance measure as default in order to find the k-nearest neighbours to your new and unknown instance. Here, the k parameter is one that you set yourself. New instances are classified by looking at the majority vote based on the training data. In case of classification, the class with the highest score wins and the unknown instance receives the

label of that winning class. If there is an equal amount of winners, the classification happens randomly. For that reason, the k parameter is often an odd number to avoid ties in the voting scores.

To build the classifier, load the library class and invoke the `fit()` function

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knnmodel = KNeighborsClassifier(n_neighbors=7)
>>> knnmodel.fit(X_train, Y_train)
```

The resulting distances matrix can now be applied to the training data itself. The `predict()` function classifies every input object and returns the corresponding class assignments

```
>>> Y_train_pred = knnmodel.predict(X_train)
```

To examine the quality of the classifier we can calculate the confusion matrix (reference/prediction) and the accuracy

```
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[229  29]
 [ 45 197]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurray Training:", acctr)
Accurray Training: 0.852
```

To test the generalizability of the classifier, it must be applied to the test data

```
>>> Y_test_pred = knnmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[218  40]
 [ 57 184]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurray Test:", accte)
Accurray Test: 0.805611222445
```

To calculate the probabilities for the class assignments instead of assigning class labels, one can use the `predict_proba()` function

```
>>> Y_test_pred = knnmodel.predict_proba(X_test)
```

To find the optimal value for k , we try different k 's using a loop and print the results

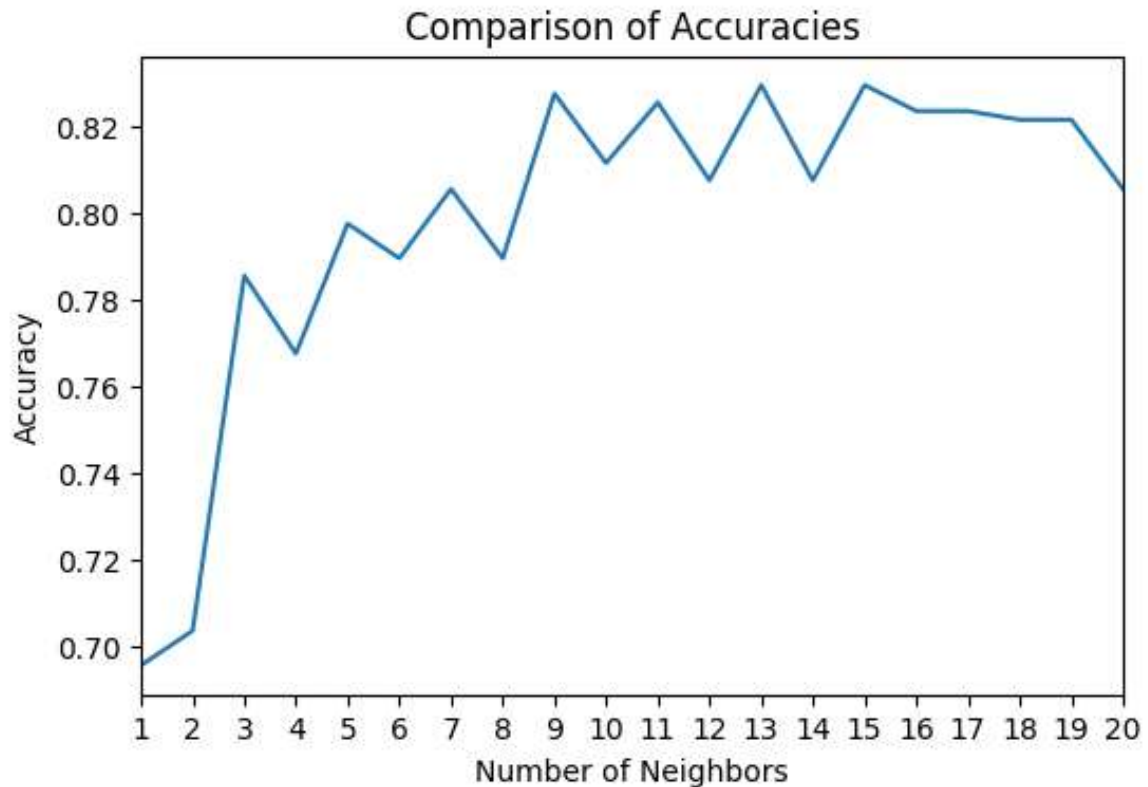
```
>>> accuracies = []
```

```
>>> for k in range(1, 21):
    knnmodel = KNeighborsClassifier(n_neighbors=k)
    knnmodel.fit(X_train, Y_train)
    Y_test_pred = knnmodel.predict(X_test)
    accte = accuracy_score(Y_test, Y_test_pred)
    print(k, accte)
    accuracies.append(accte)

1 0.695390781563
2 0.703406813627
3 0.785571142285
4 0.76753507014
5 0.797595190381
6 0.789579158317
7 0.805611222445
8 0.789579158317
9 0.827655310621
10 0.811623246493
11 0.825651302605
12 0.807615230461
13 0.829659318637
14 0.807615230461
15 0.829659318637
16 0.823647294589
17 0.823647294589
18 0.821643286573
19 0.821643286573
20 0.805611222445
```

Then, we visualize the results

```
>>> plt.plot(range(1, 21), accuracies)
>>> plt.xlim(1,20)
>>> plt.xticks(range(1, 21))
>>> plt.xlabel('Number of Neighbors')
>>> plt.ylabel('Accuracy')
>>> plt.title('Comparison of Accuracies')
>>> plt.show()
```



and choose the value of k with the highest test accuracy

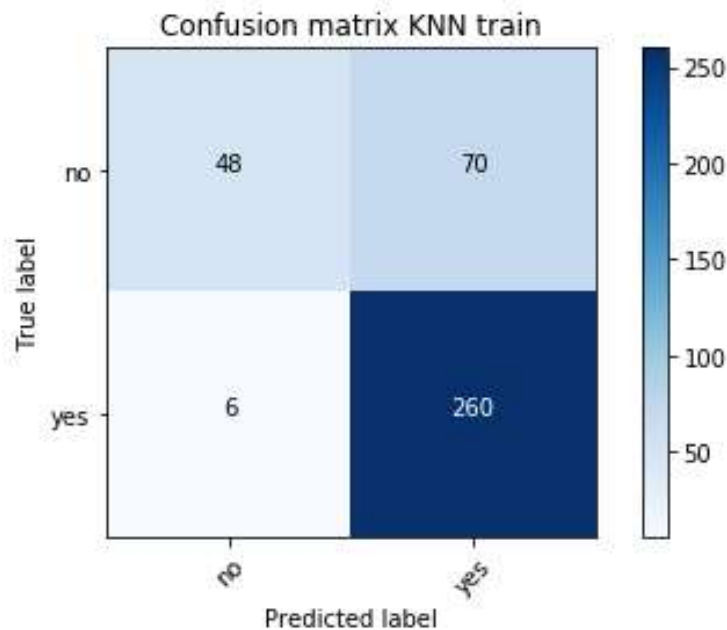
```
>>> opt_k = np.argmax(accuracies) + 1 #accuracies index starts with 0
>>> print('Optimal k =', opt_k)
Optimal k = 13
```

In our case, k=13 has the highest accuracy. Thus, we set k=13 and repeat the analysis.

For our competitive learning process, we have to store the accuracies to compare it later with the results of the other methods

```
>>> report.loc[len(report)] = ['k-NN', acctr, accte]
>>> print(report)
  Model  Acc.Train  Acc.Test
0  k-NN         0.83  0.829659
```

The Accuracy and Confusion Matrix play an important role in evaluating the results of a model. While the Accuracy is an aggregated expression for the quality of the classification, the Confusion Matrix can be used to inspect the distributions of true and false positives and negatives. This information can be used to assess whether a particular target class is better or worse classified than another. This is important if the misclassifications result in different costs, such as creditworthiness.



To calculate the **f1 measure**, the function `f1_score()` is implemented in *sklearn.metrics*. Unfortunately, this function only works if the target variable is labeled with 0 and 1. The 1's must be the positives. In our churn case, the target variable is coded with "yes" and "no". As a consequence, we have to transform it before we can use `f1_score()`. If f1 is used as the main quality measure in the complete analysis, it makes sense to transform the target variable at the beginning when defining Y. In our case, we only transform Y_test and its prediction.

To make the transformation, the `LabelEncoder` can be used (see section 2.6)

```
>>> from sklearn.preprocessing import LabelEncoder
>>> lb_churn = LabelEncoder()
>>> Y_test_code = lb_churn.fit_transform(Y_test)
>>> Y_test_pred_code = lb_churn.fit_transform(Y_test_pred)
```

Now, we can compute the f1 score

```
>>> from sklearn.metrics import f1_score
>>> flte = f1_score(Y_test_code, Y_test_pred_code)
>>> print(flte)

0.8179871520342612
```

To calculate and print the **ROC_AUC curve**, some transformations have to be made first. The ROC_AUC curve is based on probabilities. As a consequence, our model should not predict labels as outcomes but probabilities. This can be done by using the function `predict_proba()` instead of `predict`

```
>>> Y_probs = knnmodel.predict_proba(X_test)
```


The result is a numpy array containing the membership probabilities for the classes in the columns (here for class 'no' in the first column and class 'yes' in the second)

```
>>> print(Y_probs[0:6,:])

[[0.15  0.85]
 [0.31  0.69]
 [0.31  0.69]
 [0.54  0.46]
 [0.15  0.85]
 [0.46  0.54]]
```

Next, we have to transform the real output labels into a numpy array and change the labels to probabilities. This can easily done using the np.where() function

```
>>> Y_test_probs = np.array(np.where(Y_test=='yes', 1, 0))
>>> print(Y_test_probs[0:6])

[1 1 1 1 1 1]
```

We can calculate the ROC curve for a model using the function roc_curve() in scikit-learn. The function takes both the true outcomes (0,1) from the test set and the predicted probabilities for the 1 class. The function returns the false positive rates for each threshold, true positive rates for each threshold and thresholds

```
>>> from sklearn.metrics import roc_curve
>>> fpr, tpr, threshold = roc_curve(Y_test_probs, Y_probs[:, 1])
>>> print (fpr, tpr, threshold)

[0.  0.01 0.01 0.03 0.05 0.07 0.1  0.14 0.19 0.28 0.45 0.62 0.87 0.98 1.]
[0.  0.06 0.15 0.27 0.42 0.56 0.66 0.79 0.87 0.89 0.92 0.96 0.98 1. 1.]
[2.  1.   0.92 0.85 0.77 0.69 0.62 0.54 0.46 0.38 0.31 0.23 0.15 0.08 0.]
```

The AUC can be computed using the fpr and tpr previously calculated

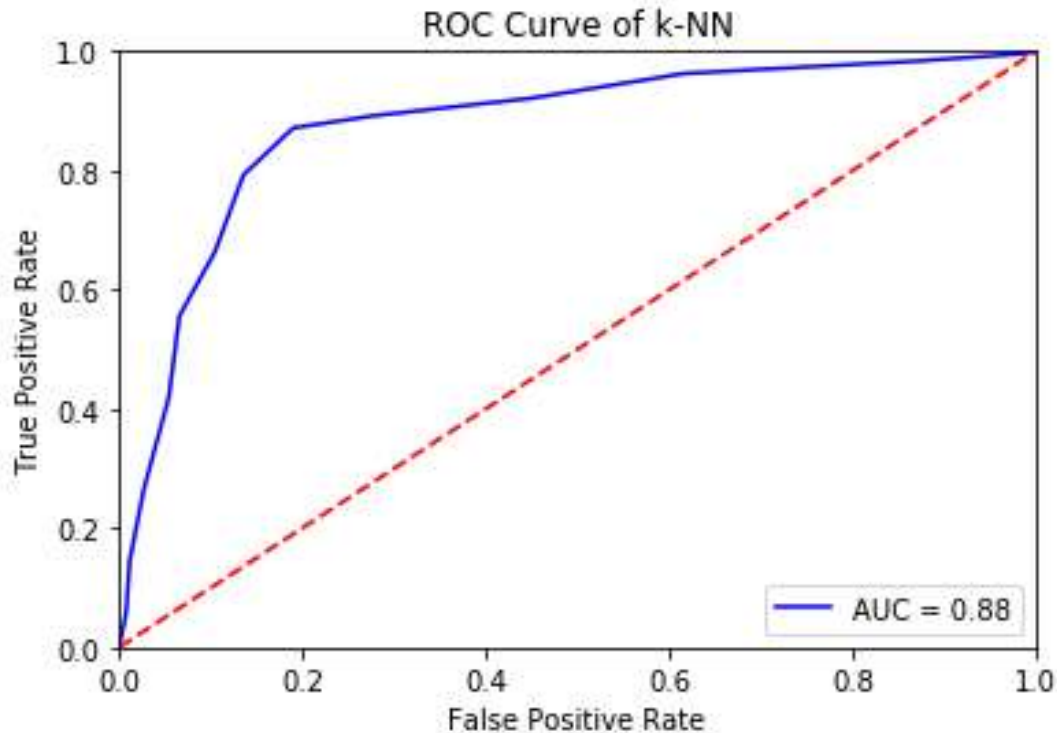
```
>>> from sklearn.metrics import auc
>>> roc_auc = auc(fpr, tpr)
>>> print(roc_auc)

0.8760252822541736
```

Now, we can plot the curve

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
>>> plt.legend(loc = 'lower right')
>>> plt.plot([0, 1], [0, 1], 'r--')
>>> plt.xlim([0, 1])
>>> plt.ylim([0, 1])
```

```
>>> plt.ylabel('True Positive Rate')
>>> plt.xlabel('False Positive Rate')
>>> plt.title('ROC Curve of k-NN')
>>> plt.show()
```



3.2 Naïve Bayes

The Naïve Bayes method is implemented in *sklearn.naive_bayes.GaussianNB*. The model will be created via

```
>>> from sklearn.naive_bayes import GaussianNB
>>> nbmodel = GaussianNB()
>>> nbmodel.fit(X_train, Y_train)
```

The model can now be applied to the training or test set via

```
>>> Y_train_pred = nbmodel.predict(X_train)
>>> Y_test_pred = nbmodel.predict(X_test)
```

The confusion matrices and the accuracies can be calculated via

```
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[209  49]
 [ 61 181]]
```

```
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurray Training:", acctr)
Accurray Training: 0.78
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[205  53]
 [ 52 189]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurray Test:", accte)
Accurray Test: 0.789579158317
```

Finally, we add the results as a new row to the report dataframe

```
>>> report.loc[len(report)] = ['Naive Bayes', acctr, accte]
```

3.3 Decision Trees

3.3.1 Simple Decision Trees

C5.0 or CART are not originally implemented but instead an information gain based algorithm where the measure of impurity can be chosen. The module *sklearn.tree.DecisionTreeClassifier* supports both Entropy and Gini Index. By default, it takes Gini.

The Decision Tree model is built via

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> etmodel = DecisionTreeClassifier(criterion='entropy', random_state=0)
>>> etmodel.fit(X_train, Y_train)
```

The parameter *criterion* sets the measure of impurity to use and the parameter *random_state* sets if there should be a seed in the random processes. If you set *random_state=0* (or any other integer value), then every time, you'll get the same result for the specific seed.

The evaluation of the quality of the model is calculated as usual

```
>>> Y_train_pred = etmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[258   0]
 [  0 242]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurray Training:", acctr)
```

```

Accurarray Training: 1.0
>>> Y_test_pred = etmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[201  57]
 [ 58 183]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurarray Test:", accte)
Accurarray Test: 0.769539078156

```

Decision trees have a tendency to overfitting. Comparing both accuracies this is clearly the case here. To reduce this overfitting effect, the modul `DecisionTreeClassifier` provides a parameter *max_depth*, which determines the maximum depth of a branch in the tree to create. This can be interpreted as a kind of prepruning.

To find the appropriate *max_depth* value, different values can be tried using a loop

```

>>> accuracies = np.zeros((2,20), float)
>>> for k in range(0, 20):
    etmodel = DecisionTreeClassifier(criterion='entropy',
                                     random_state=0, max_depth=k+1)
    etmodel.fit(X_train, Y_train)
    Y_train_pred = etmodel.predict(X_train)
    acctr = accuracy_score(Y_train, Y_train_pred)
    accuracies[0,k] = acctr
    Y_test_pred = etmodel.predict(X_test)
    accte = accuracy_score(Y_test, Y_test_pred)
    accuracies[1,k] = accte

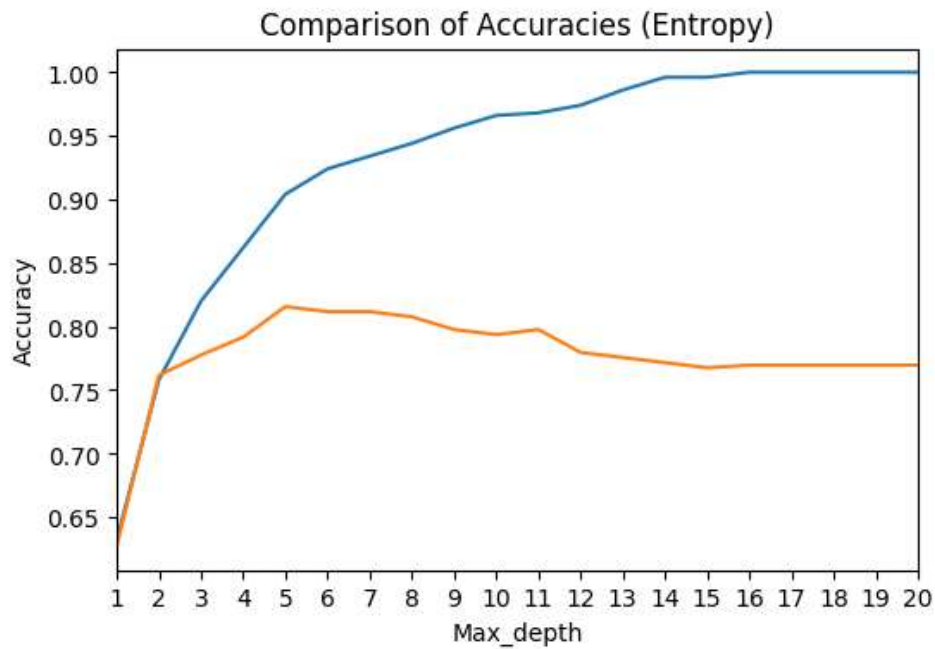
```

and compared via a line plot

```

>>> plt.plot(range(1, 21), accuracies[0,:])
>>> plt.plot(range(1, 21), accuracies[1,:])
>>> plt.xlim(1,20)
>>> plt.xticks(range(1, 21))
>>> plt.xlabel('Max_depth')
>>> plt.ylabel('Accuracy')
>>> plt.title('Comparison of Accuracies (Entropy)')
>>> plt.show()

```



In this case a max_depth of 5 seems to be appropriate. Using this parameter value, we build the tree model via

```
>>> etmodel = DecisionTreeClassifier(criterion='entropy',random_state=0,
                                     max_depth=5)

>>> etmodel.fit(X_train, Y_train)
>>> Y_train_pred = etmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[247  11]
 [ 37 205]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurray Training:", acctr)
Accurray Training: 0.904
>>> Y_test_pred = etmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[228  30]
 [ 62 179]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurray Test:", accte)
Accurray Test: 0.815631262525
>>> report.loc[len(report)] = ['Tree (Entropy)', acctr, accte]
```

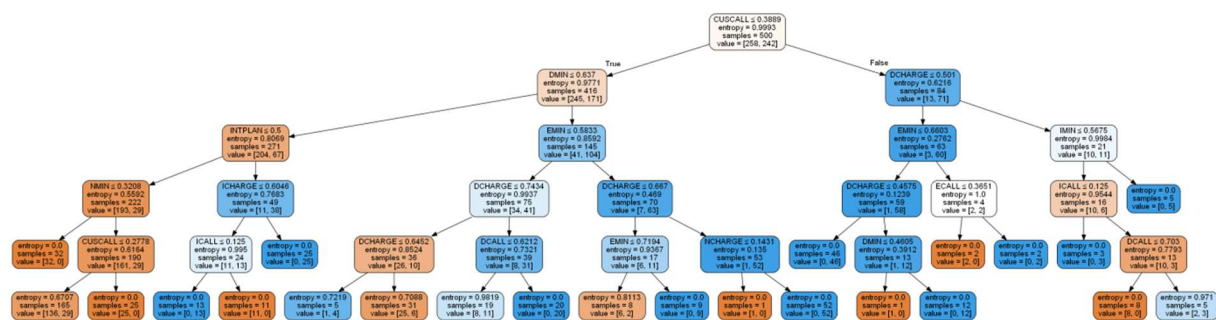
The tree can be visualized using the Graphviz package.

Graphviz is an open source visualization software library. It is designed to represent structural information as diagrams of abstract graphs and networks. To use it, the software itself and the Python interface must be installed. In an Anaconda environment this can be done using "`conda install -c conda-forge python-graphviz`".

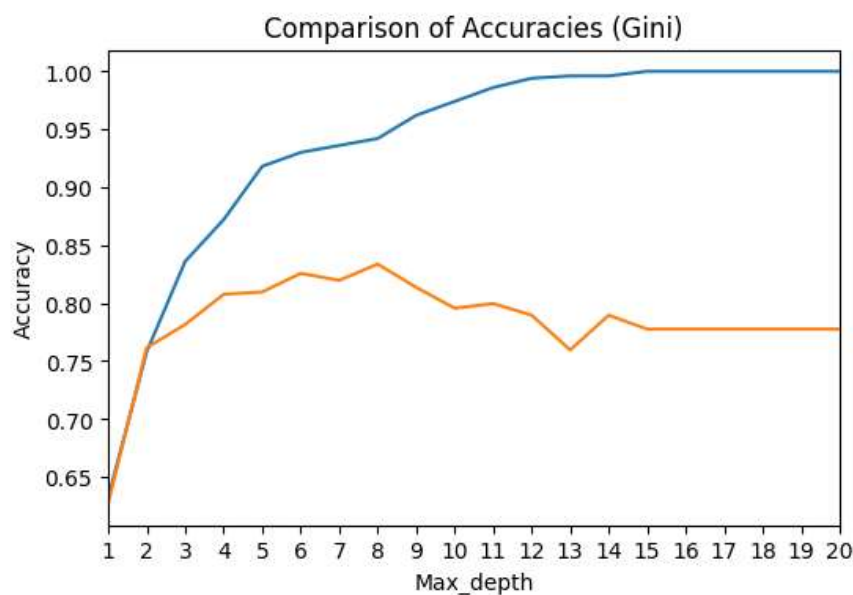
Applying the code

```
>>> import graphviz
>>> dot_data = sk.tree.export_graphviz(etmodel, out_file=None,
                                     feature_names=list(X), filled=True,
                                     rounded=True, special_characters=True)
>>> graph = graphviz.Source(dot_data)
>>> graph.format = 'png'
>>> graph.render("Churn")
```

creates a PNG file named Churn.png containing the tree of the model



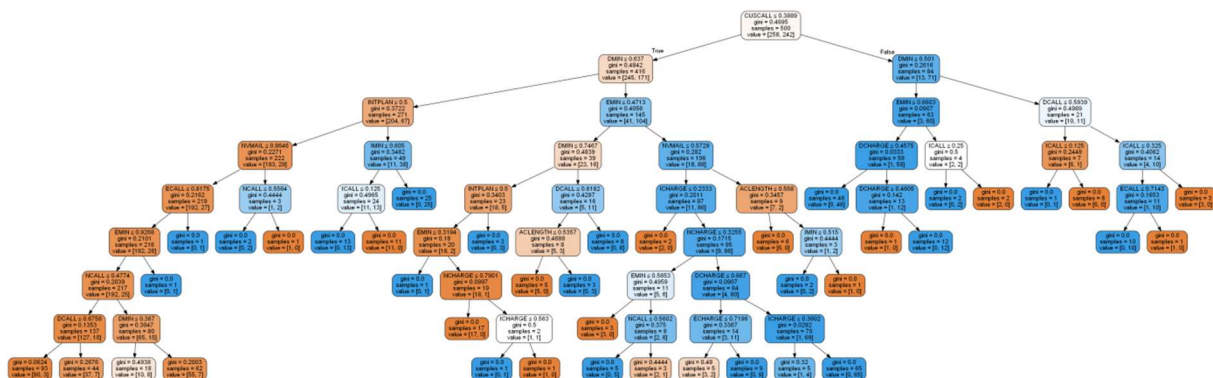
Repeating the process using the Gini index is similar. In this case only the criterion parameter has to be changed to 'gini' or left because Gini is the default measure. The resulting accuracy graph is



Using a max_depth of 8 results in

```
>>> gtmodel = DecisionTreeClassifier(random_state=0, max_depth=8)
>>> gtmodel.fit(X_train, Y_train)
>>> Y_train_pred = gtmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[257   1]
 [ 28 214]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
Accuracy Training: 0.942
>>> print("Accuracy Training:", acctr)
>>> Y_test_pred = gtmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[231  27]
 [ 56 185]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accuracy Test:", accte)
Accuracy Test: 0.833667334669
>>> report.loc[len(report)] = ['Tree (Gini)', acctr, accte]
```

and the tree has the following form



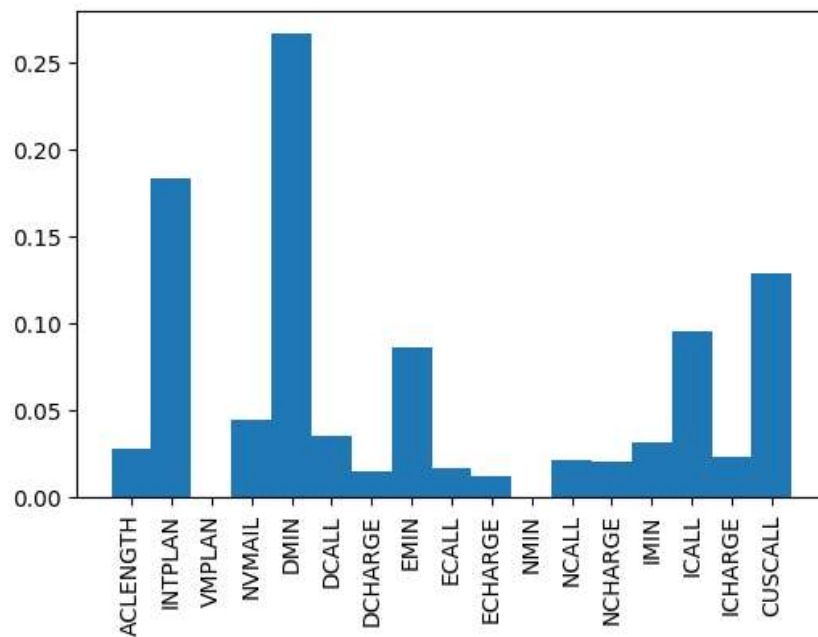
A DecisionTreeClassifier model object also contains an array `feature_importances_`. This array contains values for the importances of the feature for the classification. The higher, the more important the feature. The importance of a feature is computed here as the (normalized) total reduction of the criterion brought by that feature. It can be printed using

```
>>> list(zip(X, gtmodel.feature_importances_))
[('ACLENGTH', 0.026896119722305045),
 ('INTPLAN', 0.18300746431391479),
```

```
( 'VMPLAN', 0.0),
( 'NVMAIL', 0.043878983522002243),
( 'DMIN', 0.26642634094460632),
( 'DCALL', 0.034709388615602181),
( 'DCHARGE', 0.014107577182339114),
( 'EMIN', 0.085254260619237937),
( 'ECALL', 0.016360594391540167),
( 'ECHARGE', 0.011260457302260093),
( 'NMIN', 0.0),
( 'NCALL', 0.021041344354605807),
( 'NCHARGE', 0.020026350934531228),
( 'IMIN', 0.031519019594583093),
( 'ICALL', 0.095011424946071274),
( 'ICHARGE', 0.022293567764565113),
( 'CUSCALL', 0.12820710579183558)]
```

or it can be shown as histogram using the following code

```
>>> index = np.arange(len(gtmodel.feature_importances_))
>>> bar_width = 1.0
>>> plt.bar(index, gtmodel.feature_importances_, bar_width)
>>> plt.xticks(index, list(X), rotation=90) # labels get centered
>>> plt.show()
```

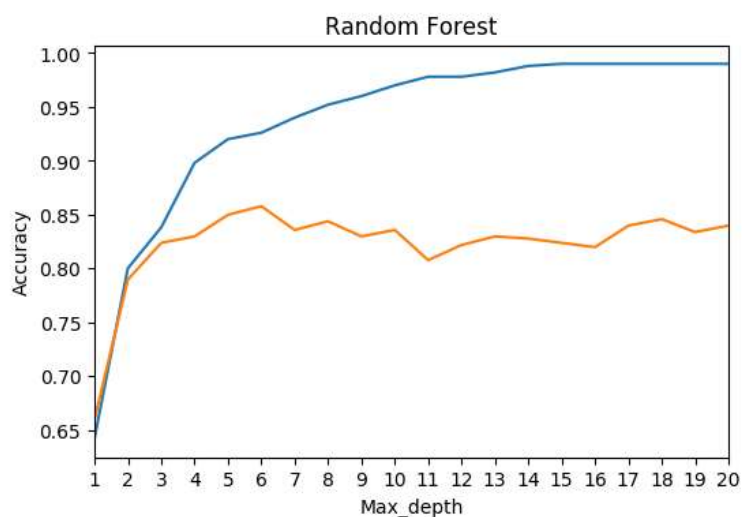


3.3.2 Random Forests

An implementation of the random forest algorithm can be found in *sklearn.ensemble.RandomForestClassifier*. The procedure to create a classifier is the same as before

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> rfmodel = RandomForestClassifier(random_state=0)
>>> rfmodel.fit(X_train, Y_train)
>>> Y_train_pred = rfmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[258  0]
 [ 5 237]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accuracy Training:", acctr)
Accuracy Training: 0.99
>>> Y_test_pred = rfmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[233  25]
 [ 55 186]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accuracy Test:", accte)
Accuracy Test: 0.839679358717
```

Important tuning parameters are here again *max_depth* and additionally *n_estimators* as the number of trees in the forest (default=10). Varying *max_depth* results in



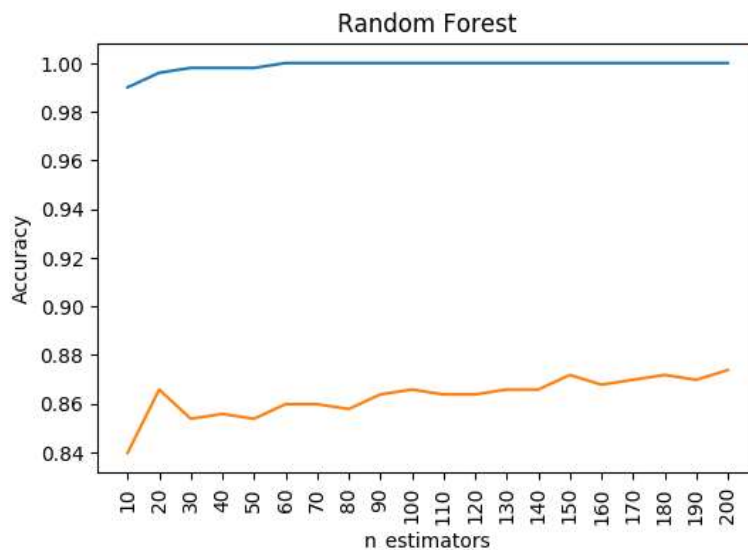
and varying `n_estimators` results in

```
>>> accuracies = np.zeros((2,20), float)
>>> ntrees = (np.arange(20)+1)*10
>>> for k in range(0, 20):
    rfmodel = RandomForestClassifier(random_state=0,
                                     n_estimators=ntrees[k])

    rfmodel.fit(X_train, Y_train)
    Y_train_pred = rfmodel.predict(X_train)
    acctr = accuracy_score(Y_train, Y_train_pred)
    accuracies[0,k] = acctr

    Y_test_pred = rfmodel.predict(X_test)
    accte = accuracy_score(Y_test, Y_test_pred)
    accuracies[1,k] = accte

>>> plt.plot(ntrees, accuracies[0,:])
>>> plt.plot(ntrees, accuracies[1,:])
>>> plt.xticks(ntrees, rotation=90)
>>> plt.xlabel('n_estimators')
>>> plt.ylabel('Accuracy')
>>> plt.title('Random Forest')
>>> plt.show()
```



One can see that both parameters have an impact on the model quality. Thus, it would be interesting if specific combinations of both parameters outperform others. To test this, we vary both parameters in the next experiment. This is called grid search. Because of the combinatorial explosion, the grid should not have too many fields. A good strategy might be to firstly perform a rough search and then build a more detailed grid around the most promising area of the first grid.

In our case, we just perform one step based on some information gathered from the isolated variations of the parameters. To store the results, we use a Numpy array having 4 columns, two for the parameter values and two for the corresponding accuracies. We perform 5 variations for max-depth and 20 for n_estimators. Thus, the result array has 100 rows. The code is as follows

```
>>> mdepth = np.linspace(4, 8, 5)
>>> accuracies = np.zeros((4,5*20), float)
>>> row = 0
>>> for k in range(0, 5):
    for l in range(0, 20):
        rfmodel = RandomForestClassifier(random_state=0,
                                         max_depth=mdepth[k], n_estimators=ntrees[l])
        rfmodel.fit(X_train, Y_train)
        Y_train_pred = rfmodel.predict(X_train)
        acctr = accuracy_score(Y_train, Y_train_pred)
        accuracies[2,row] = acctr
        Y_test_pred = rfmodel.predict(X_test)
        accte = accuracy_score(Y_test, Y_test_pred)
        accuracies[3,row] = accte
        accuracies[0,row] = mdepth[k]
        accuracies[1,row] = ntrees[l]
        row = row + 1
```

To beautify the presentations of the results, the library tabulate can be used

```
>>> from tabulate import tabulate
>>> headers = ["Max_Depth", "n_Estimators", "acctr", "accte"]
>>> table = tabulate(accuracies.transpose(), headers, tablefmt="plain",
                    floatfmt=".3f")
>>> print("\n",table)
    Max_Depth    n_Estimators    acctr    accte
      4.000         10.000    0.898    0.830
      4.000         20.000    0.908    0.842
    ...
      8.000        190.000    0.956    0.860
      8.000        200.000    0.956    0.858
```

We can now search for the highest test accuracy

```
>>> print(accuracies[3].max())
0.87374749498997994
```

Sometimes the maximal values occur twice or multiple, so that it makes sense to assign the results into an array

```
>>> maxi = np.array(np.where(accuracies==accuracies[3].max()))
```

In this case, there exists just one maximum in column 3 and row 65

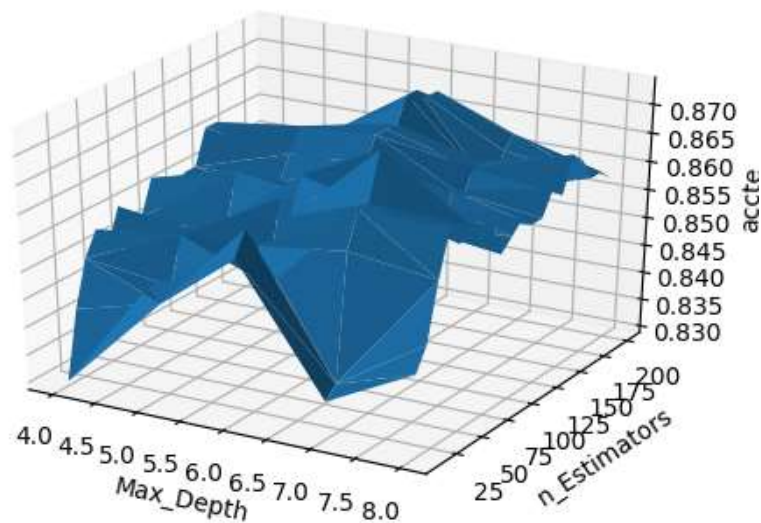
```
>>> print(maxi[0,:], maxi[1,:])
[3] [65]
```

We now use the row number to retrieve the values from the accuracies array

```
>>> table = tabulate(accuracies[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".3f")
>>> print("\n",table)
Max_Depth      n_Estimators      acctr      accte
      7.000           60.000      0.942      0.874
```

To get an impression about the relationships between the parameters and the test accuracy, we can create a surface plot

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> x = accuracies[0,:]
>>> y = accuracies[1,:]
>>> z = accuracies[3,:]
>>> ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)
>>> ax.set_xlabel('Max_Depth')
>>> ax.set_ylabel('n_Estimators')
>>> ax.set_zlabel('accte')
>>> plt.show()
```



Finally, we create the model using the optimized parameters

```
>>> rfmodel = RandomForestClassifier(random_state=0 max_depth=7,
                                     n_estimators=60)

>>> rfmodel.fit(X_train, Y_train)
>>> Y_train_pred = rfmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[256   2]
 [ 27 215]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurray Training:", acctr)
Accurray Training: 0.942
>>> Y_test_pred = rfmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[231  27]
 [ 36 205]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurray Test:", accte)
Accurray Test: 0.87374749499
>>> report.loc[len(report)] = ['Random Forest', acctr, accte]
```

3.3.3 Gradient Boosted Trees

The gradient boosting algorithm for classification is implemented in *GradientBoostingClassifier*. Using the default parameter values, it can be applied via

```
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gbmodel = GradientBoostingClassifier(random_state=0)
>>> gbmodel.fit(X_train, Y_train)
>>> Y_train_pred = gbmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[254   4]
 [ 20 222]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
```

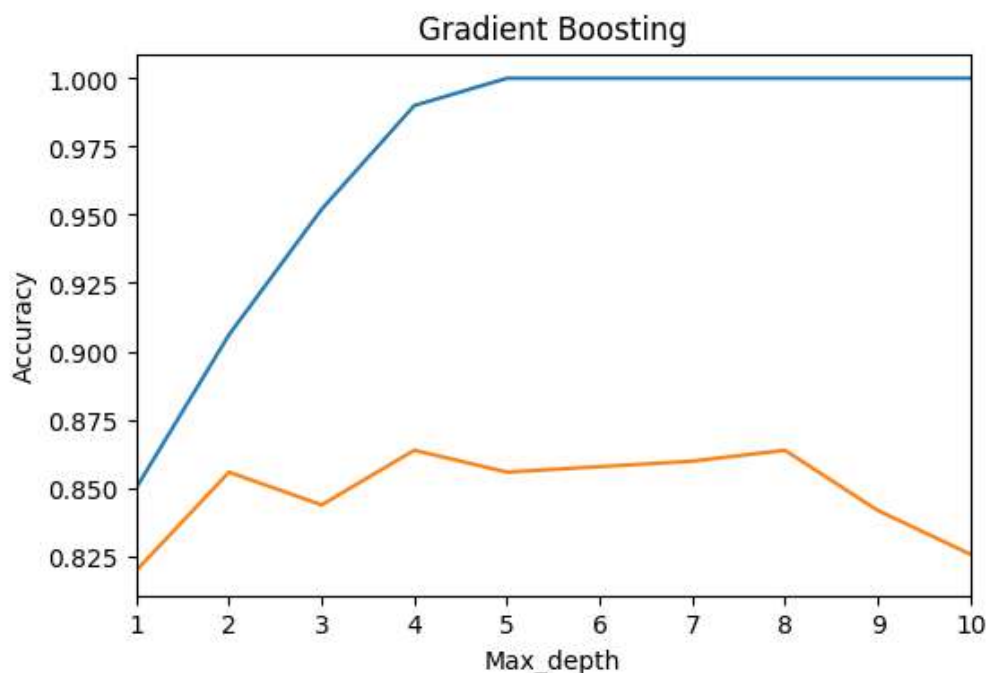
```

>>> print("Accurray Training:", acctr)
Accurray Training: 0.952
>>> Y_test_pred = gbmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[224  34]
 [ 44 197]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurray Test:", accte)
Accurray Test: 0.843687374749

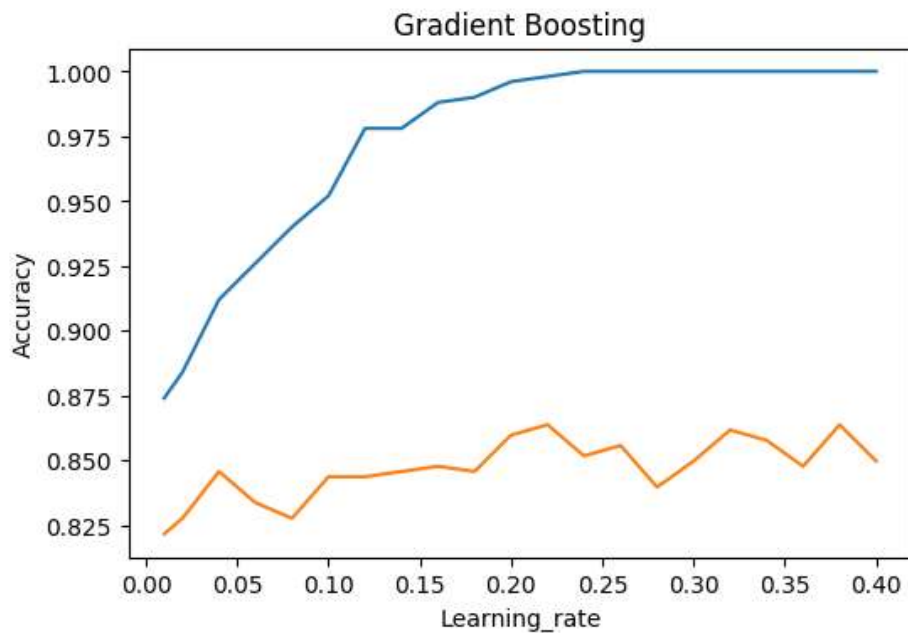
```

There are two types of parameters that can be used for tuning here: tree based and boosting parameters. There exist no general optimal values for these parameters. Thus, they must be found by trying. **Boosting parameters** are for example *learning_rate* (default=0.1) shrinking the contribution of each tree or *n_estimators* (default=100) which sets the number of boosting stages to perform. **Tree based parameters** are for example *max_depth* (default=3) which sets the maximum depth of the individual estimators, *subsample* (default=1.0) which sets the fraction of samples to be used for fitting the individual base learners or *max_features* (default=None) which sets the number of features to consider when looking for the best split. A guide about tuning Gradient Boosting in Scikit-learn can be found here: <http://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python>

Varying *max_depth* results in



and varying *learning_rate* results in



Varying *max_depth* and *learning_rate* simultaneously

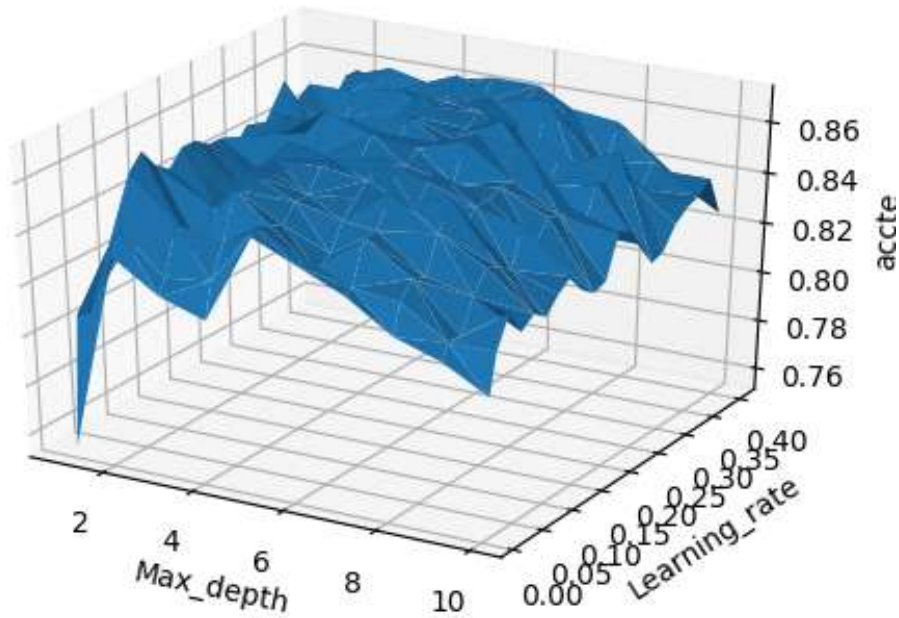
```
>>> accuracies = np.zeros((4,21*10), float)
>>> lr = np.linspace(0, 0.4, 21)
>>> lr[0] = 0.01
>>> row = 0
>>> for k in range(0, 10):
    for l in range(0, 21):
        gbmodel = GradientBoostingClassifier(random_state=0,
                                              max_depth=k+1, learning_rate=lr[l])
        gbmodel.fit(X_train, Y_train)
        Y_train_pred = gbmodel.predict(X_train)
        acctr = accuracy_score(Y_train, Y_train_pred)
        accuracies[2,row] = acctr
        Y_test_pred = gbmodel.predict(X_test)
        accte = accuracy_score(Y_test, Y_test_pred)
        accuracies[3,row] = accte
        accuracies[0,row] = k+1
        accuracies[1,row] = lr[l]
    row = row + 1
```

results in three maxima in the accuracy surface

```
>>> from tabulate import tabulate
>>> headers = ["Max_depth", "Learning_rate", "acctr", "accte"]
>>> table = tabulate(accuracies[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".3f")
```

```
>>> print("\n",table)
```

Max_depth	Learning_rate	acctr	accte
4.000	0.200	1.000	0.872
5.000	0.280	1.000	0.872
5.000	0.340	1.000	0.872



Finally, we create the model using the optimized parameters *max_depth=4* and *learning_rate=0.2* and add the results to the results dataframe.

3.4 Discriminant Analysis

The *sklearn.discriminant_analysis* module contains objects for performing linear and quadratic discriminant analysis. A linear discriminant analysis can be performed using the *LinearDiscriminantAnalysis* class and a quadratic discriminant analysis using the *QuadraticDiscriminantAnalysis* class.

Example of a linear discriminant analysis

```
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
>>> dismodel = LinearDiscriminantAnalysis()
>>> dismodel.fit(X_train, Y_train)
>>> Y_train_pred = dismodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[194  64]
 [ 65 177]]
```



```

>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurarray Training:", acctr)
Accurarray Training: 0.742
>>> Y_test_pred = dismodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
Confusion Matrix Testing:
[[192  66]
 [ 60 181]]
>>> print("Confusion Matrix Testing:\n", cmte)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurarray Test:", accte)
Accurarray Test: 0.74749498998
>>> report.loc[len(report)] = ['Linear Discriminant Analysis', acctr, accte]

```

Applying the quadratic discriminant analysis leads to the following results

```

>>> from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
>>> qdismodel = QuadraticDiscriminantAnalysis ()
>>> qdismodel.fit(X_train, Y_train)
>>> Y_train_pred = qdismodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[216  42]
 [ 36 206]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurarray Training:", acctr)
Accurarray Training: 0.844
>>> Y_test_pred = qdismodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[195  63]
 [ 46 195]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurarray Test:", accte)
Accurarray Test: 0.781563126253
>>> report.loc[len(report)] = [QuadraticDiscriminant Analysis', acctr, accte]

```

3.5 Logistic Regression

The logistic regression is implemented in *sklearn.linear_model.LogisticRegression*

```
>>> from sklearn.linear_model import LogisticRegression
>>> lrmodel = LogisticRegression()
>>> lrmodel.fit(X_train, Y_train)
```

The model can now be applied to the training or test set via

```
>>> Y_train_pred = lrmodel.predict(X_train)
>>> Y_test_pred = lrmodel.predict(X_test)
```

The confusion matrices and the accuracies can be calculated via

```
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[199  59]
 [ 72 170]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurray Training:", acctr)
Accurray Training: 0.738
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[189  69]
 [ 65 176]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurray Test:", accte)
Accurray Test: 0.731462925852
```

Finally, we add the results as a new row to the report dataframe

```
>>> report.loc[len(report)] = ['Logistic Regression', acctr, accte]
```

3.6 Neural Networks

The *sklearn.neural_network* module contains the classes to build neural networks. A neural network to build a classifier can be built via *MLPClassifier*. Important parameters are *solver* and *hidden_layer_sizes*. The parameter *solver* specifies the used solver engine for weight optimization. While 'lbfgs' is an optimizer in the family of quasi-Newton methods, 'sgd' refers to stochastic gradient descent. Usually, 'lbfgs' produces better results. The parameter *hidden_layer_sizes* specifies the number of hidden layers and their sizes. The i^{th} element

represents the number of neurons in the i^{th} hidden layer. "hidden_layer_sizes=(10,)" for example represents one hidden layer with 10 hidden neurons.

To build a neural network model, we use the following code

```
>>> from sklearn.neural_network import MLPClassifier
>>> nnetmodel = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(17,),
                             random_state=0)
>>> nnetmodel.fit(X_train, Y_train)
>>> Y_train_pred = nnetmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[243  15]
 [ 31 211]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurarray Training:", acctr)
Accurarray Training: 0.882
>>> Y_test_pred = nnetmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[220  38]
 [ 45 196]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurarray Test:", accte)
Accurarray Test: 0.811623246493
```

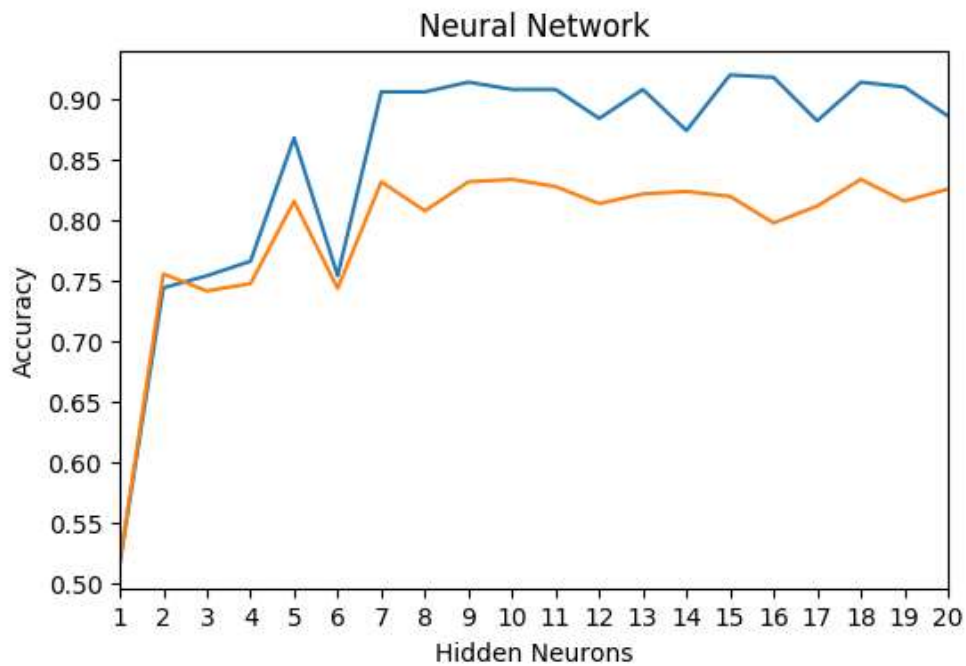
To tune the model, we vary the hidden_layer_sizes

```
>>> accuracies = np.zeros((2,20), float)
>>> for k in range(0, 20):
    nnetmodel = MLPClassifier(solver='lbfgs',
                              hidden_layer_sizes=(k+1,), random_state=0)
    nnetmodel.fit(X_train, Y_train)
    Y_train_pred = nnetmodel.predict(X_train)
    acctr = accuracy_score(Y_train, Y_train_pred)
    accuracies[0,k] = acctr
    Y_test_pred = nnetmodel.predict(X_test)
    accte = accuracy_score(Y_test, Y_test_pred)
    accuracies[1,k] = accte
>>> plt.plot(range(1, 21), accuracies[0,:])
```

```

>>> plt.plot(range(1, 21), accuracies[1,:])
>>> plt.xlim(1,20)
>>> plt.xticks(range(1, 21))
>>> plt.xlabel('Hidden Neurons')
>>> plt.ylabel('Accuracy')
>>> plt.title('Comparison of Accuracies (Neural Network)')
>>> plt.show()

```



We can obtain the accuracies array by

```

>>> headers = ["Hidden Neurons", "acctr", "accte"]
>>> table = tabulate(accuracies.transpose(), headers, tablefmt="plain",
                    floatfmt=".3f")
>>> print("\n",table)

```

Hidden Neurons	acctr	accte
1.000	0.516	0.517
2.000	0.744	0.756
3.000	0.754	0.741
4.000	0.766	0.747
5.000	0.868	0.816
6.000	0.754	0.743
7.000	0.906	0.832
8.000	0.906	0.808
9.000	0.914	0.832
10.000	0.908	0.834
11.000	0.908	0.828

12.000	0.884	0.814
13.000	0.908	0.822
14.000	0.874	0.824
15.000	0.920	0.820
16.000	0.918	0.798
17.000	0.882	0.812
18.000	0.914	0.834
19.000	0.910	0.816
20.000	0.886	0.826

and search for the maximal test accuracies with

```
>>> maxi = np.array(np.where(accuracies==accuracies[2].max()))
>>> table = tabulate(accuracies[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".3f")

>>> print("\n",table)
Hidden Neurons    acctr    accte
          10.000    0.908    0.834
          18.000    0.914    0.834
```

In this case two options result in the same test accuracy. In this case, we choose the model with the lower complexity having 10 hidden neurons.

3.7 Support Vector Machine

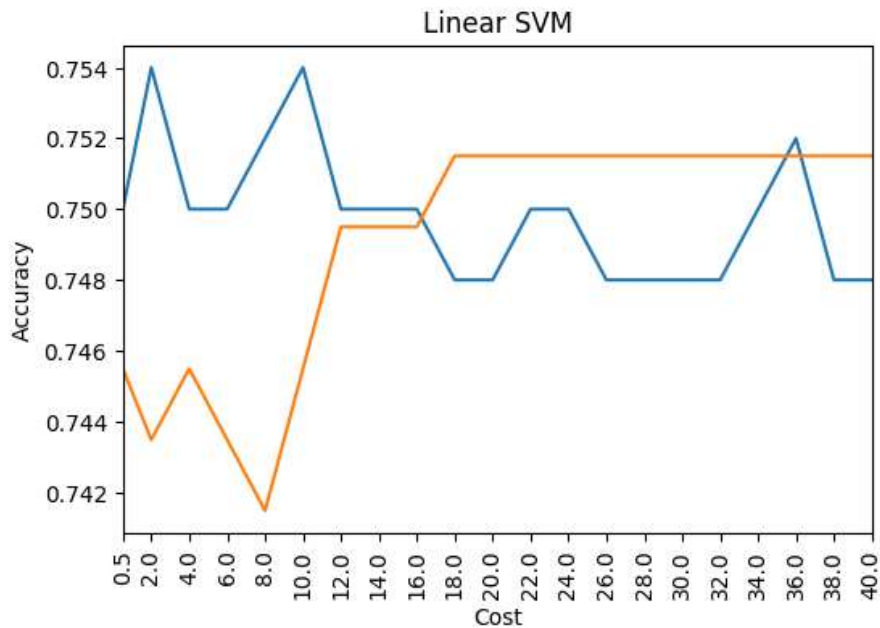
Support Vector Machines are realized in the *sklearn.svm.SVC*. Depending on the chosen kernel, different parameters can be set. Starting with the linear kernel, the *cost* parameter has to be set

```
>>> from sklearn.svm import SVC
>>> LinSVCmodel = SVC(kernel='linear', C=10, random_state=0)
>>> LinSVCmodel.fit(X_train, Y_train)
>>> Y_train_pred = LinSVCmodel.predict(X_train)
>>> cmtr = confusion_matrix(Y_train, Y_train_pred)
>>> print("Confusion Matrix Training:\n", cmtr)
Confusion Matrix Training:
[[195  63]
 [ 60 182]]
>>> acctr = accuracy_score(Y_train, Y_train_pred)
>>> print("Accurray Training:", acctr)
Accurray Training: 0.754
>>> Y_test_pred = LinSVCmodel.predict(X_test)
```

```
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[190  68]
 [ 59 182]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print("Accurray Test:", accte)
Accurray Test: 0.745490981964
```

We tune the cost parameter via

```
>>> accuracies = np.zeros((3,21), float)
>>> costs = np.linspace(0, 40, 21)
>>> costs[0] = 0.5
>>> for k in range(0, 21):
    LinSVCmodel = SVC(kernel='linear', C=costs[k], random_state=0)
    LinSVCmodel.fit(X_train, Y_train)
    Y_train_pred = LinSVCmodel.predict(X_train)
    acctr = accuracy_score(Y_train, Y_train_pred)
    accuracies[1,k] = acctr
    Y_test_pred = LinSVCmodel.predict(X_test)
    accte = accuracy_score(Y_test, Y_test_pred)
    accuracies[2,k] = accte
    accuracies[0,k] = costs[k]
>>> plt.plot(costs, accuracies[1,:])
>>> plt.plot(costs, accuracies[2,:])
>>> plt.xlim(1,20)
>>> plt.xticks(costs, rotation=90)
>>> plt.xlabel('Cost')
>>> plt.ylabel('Accuracy')
>>> plt.title('Linear SVM')
>>> plt.show()
```



When using the radial kernel, the parameter *gamma* has to be set additionally. In this case, we need to perform a grid search

```
>>> n = 21
>>> accuracies = np.zeros((4,n*n), float)
>>> costs = np.linspace(0, 20, n)
>>> costs[0] = 0.5
>>> gammas = np.linspace(0, 4.0, n)
>>> gammas[0] = 0.1
>>> row = 0
>>> for k in range(0, n):
    for l in range(0, n):
        RbfSVCmodel = SVC(kernel='rbf', C=costs[k], gamma=gammas[l],
                             random_state=0)

        RbfSVCmodel.fit(X_train, Y_train)
        Y_train_pred = RbfSVCmodel.predict(X_train)
        acctr = accuracy_score(Y_train, Y_train_pred)
        accuracies[2,row] = acctr
        Y_test_pred = RbfSVCmodel.predict(X_test)
        accte = accuracy_score(Y_test, Y_test_pred)
        accuracies[3,row] = accte
        accuracies[0,row] = costs[k]
        accuracies[1,row] = gammas[l]
        row = row + 1
```

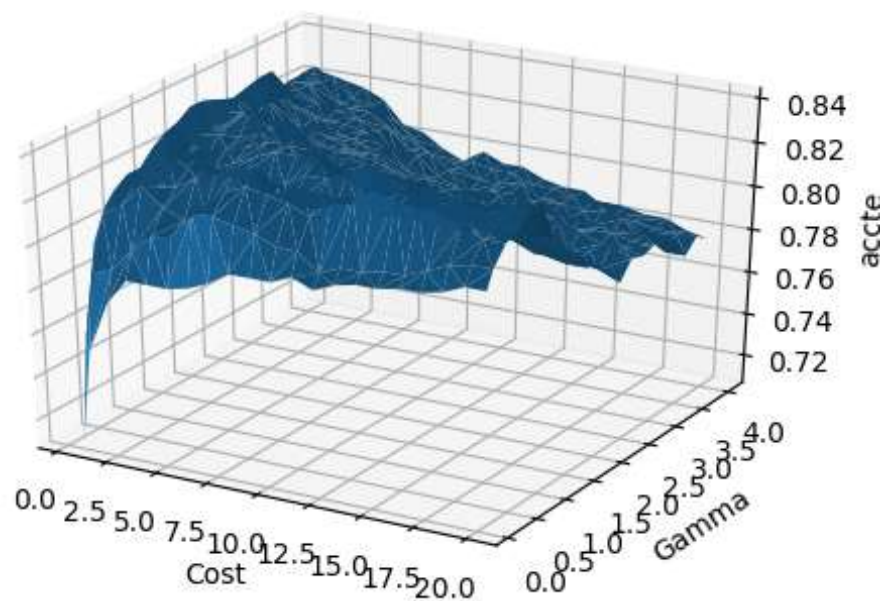
The maximum test accuracies can be obtained via

```
>>> maxi = np.array(np.where(accuracies==accuracies[3].max()))
>>> print(maxi[1,:])
>>> print(accuracies[:,maxi[1,:]])
>>> table = tabulate(accuracies[:,maxi[1,:].transpose(), headers,
                        tablefmt="plain", floatfmt=".3f")

>>> print("\n",table)
      Cost      Gamma      acctr      accte
2.000    2.600    0.932    0.842
2.000    2.800    0.938    0.842
4.000    1.800    0.938    0.842
12.000   1.000    0.938    0.842
14.000   0.400    0.898    0.842
```

We can create a surface plot via

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> x = accuracies[0,:]
>>> y = accuracies[1,:]
>>> z = accuracies[3,:]
>>> ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)
>>> ax.set_xlabel('Cost')
>>> ax.set_ylabel('Gamma')
>>> ax.set_zlabel('accte')
>>> plt.show()
```



Finally, we get

```
>>> print(report)
```

	Model	Acc.Train	Acc.Test
0	k-NN	0.830	0.829659
1	Naive Bayes	0.780	0.789579
2	Tree (Entropy)	0.904	0.815631
3	Tree (Gini)	0.942	0.833667
4	Random Forest	0.942	0.873747
5	Gradient Boosting	1.000	0.869739
6	Linear Discriminant Analysis	0.742	0.747495
7	Quadratic Discriminant Analysis	0.844	0.781563
8	Logistic Regression	0.738	0.731463
9	Neural Network	0.908	0.833667
10	SVM (Linear)	0.754	0.745491
11	SVM (Radial)	0.898	0.841683

3.8 Using Cross Validation

In our analyses before, the data we used was splitted into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset in order to test our model's prediction on this subset. We use the test dataset too to adjust the parameters of the algorithms in order to optimize the generalizability of our models.

But train/test split does have its dangers. What if the split we make isn't random? What if one subset of our data has only people from a certain state, employees with a certain income level but no other income levels, only women or only people at a certain age? This will result in overfitting, even though we're trying to avoid it. Additionally, if we always use the same test data for model selection and adjustment, they become part of the training data and overfitting will become more likely.

This is where cross validation comes into play. It is very similar to train/test split, but it is applied to more subsets. Applying cross validation, we split our data into k complementary subsets, and train on k-1 one of those subset. We always hold one subset for test. We are able to do it for each of the subsets. The further advantage of this procedure is that we avoid wasting training data for testing purposes.

We can apply cross validation for model selection and adjustment. Once the model type and its optimal parameters have been selected, a final model is trained using these hyperparameters on the **full** training set, and the generalization quality is measured on the test set.

To apply cross validation, one way is to call the machine learning algorithm k times, each time giving a different train/test set. You will then need to calculate the accuracy for each stage, and average it at the end. To support the data scientist, Scikit-learn provides a helper function named *cross_val_score* which performs all the necessary operations.

First, we have to load and prepare the data

```
>>> data_ori = pd.read_csv('churn_balanced.csv')
>>> X_ori = data_ori.drop('CHURN', axis = 1)
>>> Y = data_ori['CHURN']
>>> X = (X_ori-X_ori.min())/(X_ori.max()-X_ori.min())

>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                                    test_size = 0.2)

>>> report = pd.DataFrame(columns=['Model', 'Mean Acc. Training',
                                   'Standard Deviation', 'Acc. Test'])
```

Now, we apply *cross_val_score* to the first algorithm. At first, we have to instantiate the algorithm as usual

```
>>> from sklearn.linear_model import LogisticRegression
>>> lrmodel = LogisticRegression()
```

Then, we can apply *cross_val_score* to the algorithm

```
>>> from sklearn.model_selection import cross_val_score
>>> accuracies = cross_val_score(lrmodel, X_train, Y_train,
                                scoring='accuracy', cv = 10)
```

Input parameters are the model instance and the training data. The parameter *scoring* defines the scoring measure and *cv* the number of k folds of cross validation. In our case, the training set is splitted into 10 parts and for every of the 10 combinations a model is calculated and evaluated with the accuracy. Thus, the result consists of 10 accuracy values which are the scores of the 10 **test** folds (not the test sample)

```
>>> print(accuracies)
[ 0.69135802  0.7          0.7625          0.7625          0.6875          0.775
 0.725         0.7375         0.74683544  0.70886076]
```

To measure the quality of the algorithm, we compute the mean and the standard deviation of the accuracies

```
>>> acc_mean = accuracies.mean()
>>> print(acc_mean)
```

```
0.740932
>>> acc_std = accuracies.std()
>>> print(acc_std)
0.0435317
```

Finally, we have to evaluate the quality of the model applied to the real test data which was not used by the model until now

```
>>> lrmodel.fit(X_train, Y_train)
>>> Y_test_pred = lrmodel.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> print(accte)
0.765
```

The result can now be stored for later comparison in the report dataframe

```
>>> report.loc[len(report)] = ['Logistic Regression', accuracies.mean(),
                                accuracies.std(), accte]
```

To print the result in a more readable and compact way, one can use

```
>>> print(report.loc[len(report)-1])
Model                Logistic Regression
Mean Acc. Training    0.740932
Standard Deviation    0.0435317
Acc. Test             0.765
```

We repeat the procedure now for the Naïve Bayes algorithm

```
>>> from sklearn.naive_bayes import GaussianNB
>>> nbmodel = GaussianNB()
>>> from sklearn.model_selection import cross_val_score
>>> accuracies = cross_val_score(nbmodel, X_train, Y_train,
                                scoring='accuracy', cv = 10)

>>> print(accuracies)
[ 0.69135802  0.725      0.85      0.775      0.6875      0.8125
  0.8125      0.8       0.78481013  0.7721519 ]
>>> nbmodel.fit(X_train, Y_train)
>>> Y_test_pred = nbmodel.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['Naive Bayes', accuracies.mean(),
                                accuracies.std(), accte]

>>> print(report.loc[len(report)-1])
Model                Naive Bayes
Mean Acc. Training    0.792204
```

```
Standard Deviation      0.0399891
Acc. Test               0.795
Name: 1, dtype: object
```

As one can see, the results are of a slightly higher quality.

In the case of algorithms having hyperparameters, it makes sense to find the optimal parameters first and to use them for model fitting. This can be done in Scikit-learn using the *GridSearchCV* class. After predefining the grid of parameters, the function analyses the different alternatives using cross validation.

We apply the function to the random forest algorithm. At first, we instantiate the algorithm

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> rfmodel = RandomForestClassifier(random_state=0)
```

Next, we define the parameter grid and instantiate the *GridSearchCV* class

```
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {
    'max_depth': [ 4.,  5.,  6.,  7.,  8.],
    'n_estimators': [ 10,  50, 100, 150, 200]
}
>>> CV_rfmodel = GridSearchCV(estimator=rfmodel, param_grid=param_grid,
                               cv=10)
```

Now, we use the fit method from the object which starts the calculation of all models. This step is very calculation intensive since there might be many models to calculate (in our case $5 \times 5 \times 10 = 250$)

```
>>> CV_rfmodel.fit(X_train, Y_train)
```

Next, we display the best parameters

```
>>> print(CV_rfmodel.best_params_)
{'max_depth': 8.0, 'n_estimators': 150}
```

These parameters can now be used to finally fit the model using the complete training data set

```
>>> rfmodel = rfmodel.set_params(**CV_rfmodel.best_params_)
>>> rfmodel.fit(X_train, Y_train)
```

Finally, we evaluate the quality of the model using the real test data set

```
>>> Y_test_pred = rfmodel.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
```

The results of the gridsearch process are stored in a data structure named `cv_results_`. It is an attribute of the model instance. Here we can find the mean accuracies and the corresponding standard deviations for the different trials (see http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html). For our report dataframe, we need both values from the model with the best parameters. The number of the corresponding row is given by the attribute `best_index_`

```
>>> report.loc[len(report)] = ['Random Forest (grid)',
                                CV_rfmodel.cv_results_['mean_test_score'][CV_rfmodel.best_index_],
                                CV_rfmodel.cv_results_['std_test_score'][CV_rfmodel.best_index_],
                                accte]

>>> print(report.loc[len(report)-1])
Model                Random Forest (grid)
Mean Acc. Training          0.878598
Standard Deviation          0.0418233
Acc. Test                   0.89
Name: 2, dtype: object
```

We repeat the procedure for the gradient boosting classifier, adapting the code to the relevant parameters:

```
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gbmodel = GradientBoostingClassifier(random_state=0)
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {
    'max_depth': [ 3., 4., 5.],
    'subsample': [0.7, 0.8, 0.9],
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.1, 0.2, 0.3]
}
>>> CV_gbmodel = GridSearchCV(estimator=gbmodel, param_grid=param_grid,
                               cv=10)
>>> CV_gbmodel.fit(X_train, Y_train)
>>> print(CV_gbmodel.best_params_)
>>> gbmodel = gbmodel.set_params(**CV_gbmodel.best_params_)
>>> gbmodel.fit(X_train, Y_train)
>>> Y_test_pred = gbmodel.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
```

```
>>> report.loc[len(report)] = ['Gradient Boosting (grid)',
    CV_gbmodel.cv_results_['mean_test_score'][CV_gbmodel.best_index_],
    CV_gbmodel.cv_results_['std_test_score'][CV_gbmodel.best_index_],
    accte]

>>> print(report.loc[len(report)-1])
Model                Gradient Boosting (grid)
Mean Acc. Training                0.87234
Standard Deviation                0.0288931
Acc. Test                        0.89
Name: 3, dtype: object
```

We can continue in the same way for the other algorithms.

Finally, the complete report can be printed

	Model	Mean Acc. Training	Standard Deviation	Acc. Test
0	Logistic Regression	0.740932	0.043532	0.765
1	Naive Bayes	0.792204	0.039989	0.795
2	Random Forest (grid)	0.878598	0.041823	0.890
3	Gradient Boosting (grid)	0.872340	0.028893	0.890

In this section we have used accuracy as a measure of quality. This is sufficient as long as the data is balanced here. If the f1 measure is to be used instead, e.g. in the case of an unbalanced data set, then this has to be set as a scoring parameter. Since `f1_score()` in *sklearn.metrics* requires a target variable in the form of 0 and 1, it must first be transformed here. For this we use the LabelEncoder.

Instead of

```
>>> Y = data_ori['CHURN']
```

we have to write

```
>>> from sklearn.preprocessing import LabelEncoder

>>> lb_churn = LabelEncoder()

>>> data_ori['CHURN_code'] = lb_churn.fit_transform(data_ori['CHURN'])

>>> data_ori[['CHURN', 'CHURN_code']].head(10)

>>> Y = data_ori['CHURN_code']
```

Now, we can build our model using the f1 measure, e.g.

```
>>> from sklearn.linear_model import LogisticRegression

>>> lrmodel_f1 = LogisticRegression()

>>> from sklearn.model_selection import cross_val_score
```

```

>>> fles = cross_val_score(lrmodel_f1, X_train, Y_train, scoring='f1',
                           cv = 10)

>>> print("fles = ", fles)

fles =  [0.69444444 0.69230769 0.68421053 0.71428571 0.69333333 0.7027027
        0.78481013 0.72222222 0.56716418 0.74074074]

>>> print("Mean = ", fles.mean())

Mean =  0.6996221682039395

>>> print("SD = ", fles.std())

SD =  0.05252189484892129

>>> lrmodel_f1.fit(X_train, Y_train)

>>> Y_test_pred = lrmodel_f1.predict(X_test)

>>> flte = f1_score(Y_test, Y_test_pred)

>>> report.loc[len(report)] = ['Logistic Regression', fles.mean(),
                              fles.std(), flte]

>>> print(report.loc[len(report)-1])

Model                Logistic Regression
Mean Acc. Training    0.699622
Standard Deviation    0.0525219
Acc. Test             0.738462
Name: 0, dtype: object

```

3.9 Cross Validation and Imbalanced Classes

To handle the case of imbalanced classes, there exist two possible ways: (1) down- or upsampling to get balanced classes or (2) apply cost-sensitive learning by using class weights. Option (1) was already mentioned in Chapter 2. Here, we want to apply option 2.

To demonstrate the usage, we first load the unbalanced version of our churn dataset

```

>>> data_ori = pd.read_excel('churn_unbalanced.xls')
>>> print(data_ori.shape)

(3333, 18)

```

Now, we prepare the data and create our training and test sample

```

>>> X_ori = data_ori.drop('CHURN', axis = 1)
>>> Y = data_ori['CHURN']
>>> X = (X_ori-X_ori.min())/(X_ori.max()-X_ori.min())

```

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                                    test_size = 0.2)

>>> print(Y_train.value_counts())
0    2278
1     388
Name: CHURN, dtype: int64
>>> print(Y_test.value_counts())
0    572
1     95
Name: CHURN, dtype: int64
>>> report = pd.DataFrame(columns=['Model', 'Mean Training',
                                'Standard Deviation', 'Test'])
```

Now, we apply Logistic Regression using the unbalanced data and accuracy as measure

```
>>> from sklearn.linear_model import LogisticRegression
>>> lrmodel = LogisticRegression()
>>> from sklearn.model_selection import cross_val_score
>>> accuracies = cross_val_score(lrmodel, X_train, Y_train,
                                scoring='accuracy', cv = 10)

>>> lrmodel.fit(X_train, Y_train)
>>> Y_test_pred = lrmodel.predict(X_test)
>>> cmte = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte)
Confusion Matrix Testing:
[[565   7]
 [ 80  15]]
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['LR unbalanced Acc', accuracies.mean(),
                              accuracies.std(), accte]

>>> print(report.loc[len(report)-1])
Model                LR unbalanced Acc
Mean Acc. Training      0.861214
Standard Deviation      0.00751363
Acc. Test                0.869565
Name: 0, dtype: object
```

The confusion matrix indicates, that this is not a desirable result. Nevertheless, the accuracy has a high value because of the good classifications of the greater class. But this is exactly not the class that is of interest.

Next, we repeat the analysis with the f1 measure

```
>>> from sklearn.linear_model import LogisticRegression
>>> lrmodel_f1 = LogisticRegression()
>>> from sklearn.model_selection import cross_val_score
>>> f1es = cross_val_score(lrmodel_f1, X_train, Y_train, scoring='f1',
                           cv = 10)

>>> lrmodel_f1.fit(X_train, Y_train)
>>> Y_test_pred = lrmodel_f1.predict(X_test)
>>> cmte_f1 = confusion_matrix(Y_test, Y_test_pred)
>>> print("Confusion Matrix Testing:\n", cmte_f1)
Confusion Matrix Testing:
[[565   7]
 [ 80  15]]
>>> f1te = f1_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['LR unbalanced f1', f1es.mean(),
                               f1es.std(), f1te]

>>> print(report.loc[len(report)-1])
Model                LR unbalanced f1
Mean Acc. Training      0.214284
Standard Deviation      0.079941
Acc. Test               0.25641
```

While the confusion matrix is still the same, the f1 scores show the extrem low quality of our classifier.

To handle imbalanced classes, sklearn provides a parameter for its classificatory with the name *class_weight*. The mode *class_weight='balanced'* uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as $n_samples / (n_classes * np.bincount(y))$.

If we use this parameter when instantiating the classificatory

```
>>> lrmodel = LogisticRegression(class_weight='balanced')
```

we get the following results

```
Confusion Matrix Testing:
[[437 135]
 [ 19  76]]

Model                LR balanced Acc
Mean Acc. Training      0.760707
Standard Deviation      0.0185798
Acc. Test               0.769115
```

The confusion matrix shows, that the result is much more balanced out between the two classes. We also see this in the f1 values

Model	LR balanced f1
Mean Acc. Training	0.4801
Standard Deviation	0.0209967
Acc. Test	0.496732

We repeat this for our Random Forest classifier, where we set

```
>>> rfmodel = RandomForestClassifier(class_weight='balanced',
                                     random_state=0)
```

After training all of the variants, we receive the following result

	Model	Mean Training	Standard Deviation	Test
0	LR unbalanced Acc	0.861214	0.007514	0.869565
1	LR unbalanced f1	0.214284	0.079941	0.256410
2	LR balanced Acc	0.760707	0.018580	0.769115
3	LR balanced f1	0.480100	0.020997	0.496732
4	RF unbalanced Acc	0.949737	0.008079	0.950525
5	RF unbalanced f1	0.804357	0.028492	0.811429
6	RF balanced Acc	0.944861	0.014038	0.934033
7	RF balanced f1	0.801308	0.047063	0.765957

Interestingly, the unbalanced variant is better than the balanced one. However, both are at a high level.

3.10 Ensemble Learning

Even if Scikit-learn has functions to build ensemble models, the corresponding functions of the library *mlex* are more powerful and more convenient to use. For building ensemble classifiers the classes *EnsembleVoteClassifier* and *StackingClassifier* can be used. Both will be the subject of the following description.

As the name suggests, the class *EnsembleVoteClassifier* offers functions for creating voting-based ensemble classifiers. It is a meta-classifier for combining similar or conceptually different machine learning classifiers for classification via majority voting. The *EnsembleVoteClassifier* implements hard and soft voting. In hard voting, the final class label is predicted as the class label that has been predicted most frequently by the classification models of the ensemble. In soft voting, the class labels are predicted by averaging the class-probabilities of the ensemble's models. To demonstrate the creation and usage of an *EnsembleVoteClassifier*, we use our churn data set.


```

>>> print(report.loc[len(report)-1])
Model                Naive Bayes
Mean Acc. Training    0.798513
Standard Deviation    0.0341217
Acc. Test             0.74
Name: 1, dtype: object

>>> from sklearn.ensemble import RandomForestClassifier
>>> rfmodel = RandomForestClassifier(random_state=0)
>>> accuracies = cross_val_score(rfmodel, X_train, Y_train,
                                scoring='accuracy', cv=10)

>>> rfmodel.fit(X_train, Y_train)
>>> Y_test_pred = rfmodel.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['Random Forest', accuracies.mean(),
                                accuracies.std(), accte]

>>> print(report.loc[len(report)-1])
Model                Random Forest
Mean Acc. Training    0.858608
Standard Deviation    0.0529722
Acc. Test             0.87
Name: 2, dtype: object

```

Now, we can create the ensemble classifiers using *EnsembleVoteClassifier*. To create an ensemble using hard voting, we need at least the *clfs* parameter, which has to be a list of the classifiers used for the ensemble

```

>>> from mlxtend.classifier import EnsembleVoteClassifier
>>> ens1model = EnsembleVoteClassifier(clfs=[knnmodel, nbmodel, rfmodel])

```

After instantiating the ensemble classifier it can be used applying *cross_val_score* as in the chapter before

```

>>> accuracies = cross_val_score(ens1model, X_train, Y_train,
                                scoring='accuracy', cv=10)

>>> ens1model.fit(X_train, Y_train)
>>> Y_test_pred = ens1model.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['Ensemble (hard)',
                                accuracies.mean(), accuracies.std(), accte]

```

```
>>> print(report.loc[len(report)-1])
Model                Ensemble (hard)
Mean Acc. Training    0.838544
Standard Deviation    0.0423817
Acc. Test              0.805
Name: 3, dtype: object
```

EnsembleVoteClassifier provides the opportunity to set weights for the models

```
>>> ens2model = EnsembleVoteClassifier(clfs=[knnmodel, nbmodel, rfmodel],
                                       weights=[1,1,2])
>>> accuracies = cross_val_score(ens2model, X_train, Y_train,
                                   scoring='accuracy', cv=10)
>>> ens2model.fit(X_train, Y_train)
>>> Y_test_pred = ens2model.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['Ens. (weighted, hard)',
                               accuracies.mean(), accuracies.std(), accte]
>>> print(report.loc[len(report)-1])
Model                Ens. (weighted, hard)
Mean Acc. Training    0.852342
Standard Deviation    0.0492377
Acc. Test              0.845
Name: 4, dtype: object
```

The *voting* parameter allows switching from hard to soft voting

```
>>> ens3model = EnsembleVoteClassifier(clfs=[knnmodel, nbmodel, rfmodel],
                                       weights=[1,1,2], voting='soft')
>>> accuracies = cross_val_score(ens3model, X_train, Y_train,
                                   scoring='accuracy', cv=10)
>>> ens3model.fit(X_train, Y_train)
>>> Y_test_pred = ens3model.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['Ens. (weighted, soft)',
                               accuracies.mean(), accuracies.std(), accte]
>>> print(report.loc[len(report)-1])
Model                Ens. (weighted, soft)
Mean Acc. Training    0.837278
Standard Deviation    0.03363
Acc. Test              0.82
Name: 5, dtype: object
```

Instead of *cross_val_score* the function *GridSearchCV* can be used to test different hyperparameters

```
>>> ens4model = EnsembleVoteClassifier(clfs=[knnmodel, nbmodel, rfmodel],
                                     weights=[1,1,2], voting='soft')

>>> from sklearn.model_selection import GridSearchCV

>>> param_grid = {
    'kneighborsclassifier__n_neighbors': [ 5, 7, 9],
    'randomforestclassifier__n_estimators': [ 100, 150, 200]
}

>>> CV_ensmodel = GridSearchCV(estimator=ens4model,
                               param_grid=param_grid, cv=10)

>>> CV_ensmodel.fit(X_train, Y_train)

>>> print('Best Parameters:', CV_ensmodel.best_params_)
Best Parameters: {'kneighborsclassifier__n_neighbors': 5,
                  'randomforestclassifier__n_estimators': 100}

>>> ens4model = ens4model.set_params(**CV_ensmodel.best_params_)

>>> ens4model.fit(X_train, Y_train)

>>> Y_test_pred = ens4model.predict(X_test)

>>> accte = accuracy_score(Y_test, Y_test_pred)

>>> report.loc[len(report)] = ['Ensemble (gridsearch)',
                              CV_ensmodel.cv_results_['mean_test_score'][CV_ensmodel.best_index_],
                              CV_ensmodel.cv_results_['std_test_score'][CV_ensmodel.best_index_],
                              accte]

>>> print(report.loc[len(report)-1])
Model                               Ensemble (gridsearch)
Mean Acc. Training                   0.846058
Standard Deviation                   0.0411706
Acc. Test                           0.835
Name: 6, dtype: object
```

Stacking is an ensemble learning technique that combines multiple classification models via a meta-classifier (supervisor). First, the individual classification models are trained based on the complete training set. Then, the meta-classifier is fitted based on the outputs of the individual classification models in the ensemble. The meta-classifier can either be trained on the predicted class labels or probabilities from the ensemble.

To apply stacking, *mlxtend* provide the class *StackingClassifier*. At first, the classifiers have to be instantiated. We use the ones from the step before. In the next step, the meta-classifier must be instantiated. In our case, we use a logistic regression

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr_ensemble = LogisticRegression()
```

Now, we instantiate the *StackingClassifier* and continue as before

```
>>> from mlxtend.classifier import StackingClassifier
>>> stens1model = StackingClassifier(classifiers=[knnmodel, nbmodel,
                                                rfmodel], use_probas=True,
                                    average_probas=False,
                                    meta_classifier=lr_ensemble)
>>> accuracies = cross_val_score(stens1model, X_train, Y_train,
                                  scoring='accuracy', cv=10)
>>> stens1model.fit(X_train, Y_train)
>>> Y_test_pred = stens1model.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['Stacking Ensemble', accuracies.mean(),
                              accuracies.std(), accte]
>>> print(report.loc[len(report)-1])
Model                Stacking Ensemble
Mean Acc. Training    0.876076
Standard Deviation    0.0381657
Acc. Test             0.855
Name: 7, dtype: object
```

Instead of *cross_val_score*, we can use the function *GridSearchCV* here too

```
>>> stens2model = StackingClassifier(classifiers=[knnmodel, nbmodel,
                                                rfmodel], use_probas=True,
                                    average_probas=False,
                                    meta_classifier=lr_ensemble)
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {
    'kneighborsclassifier__n_neighbors': [ 5,  7,  9],
    'randomforestclassifier__n_estimators': [ 100, 150, 200]
}
>>> CV_ensmodel = GridSearchCV(estimator=stens2model,
                               param_grid=param_grid, cv=10)
>>> CV_ensmodel.fit(X_train, Y_train)
>>> print('Best Parameters:', CV_ensmodel.best_params_)
Best Parameters: {'kneighborsclassifier__n_neighbors': 5,
                  'randomforestclassifier__n_estimators': 100}
>>> stens2model = stens2model.set_params(**CV_ensmodel.best_params_)
```

```
>>> stens2model.fit(X_train, Y_train)
>>> Y_test_pred = stens2model.predict(X_test)
>>> accte = accuracy_score(Y_test, Y_test_pred)
>>> report.loc[len(report)] = ['Stacking (gridsearch)',
    CV_ensmodel.cv_results_['mean_test_score'][CV_ensmodel.best_index_],
    CV_ensmodel.cv_results_['std_test_score'][CV_ensmodel.best_index_],
    accte]
>>> print(report.loc[len(report)-1])
Model                Stacking (gridsearch)
Mean Acc. Training                0.876095
Standard Deviation                0.0381857
Acc. Test                        0.855
Name: 8, dtype: object
```

Finally, we can compare all of our models

```
>>> print(report)
```

	Model	Mean Acc. Training	Standard Deviation	Acc. Test
0	k-NN	0.818513	0.047880	0.775
1	Naive Bayes	0.798513	0.034122	0.740
2	Random Forest	0.858608	0.052972	0.870
3	Ensemble (hard)	0.838544	0.042382	0.805
4	Ens. (weighted, hard)	0.852342	0.049238	0.845
5	Ens. (weighted, soft)	0.837278	0.033630	0.820
6	Ensemble (gridsearch)	0.846058	0.041171	0.835
7	Stacking Ensemble	0.876076	0.038166	0.855
8	Stacking (gridsearch)	0.876095	0.038186	0.855

3.11 Applying a Classification Model to New Data

If a model is trained on scaled data, it will only work with scaled data. Note, that the scaling must be exactly the same as for the training data. Thus, it is necessary, to keep the scaler. If, for example, the data was scaled using a sklearn scaler

```
>>> nscaler = preprocessing.MinMaxScaler()
>>> data = nscaler.fit_transform(data_ori)
```

then new data can be easily scaled using

```
>>> data = nscaler.fit_transform(newdata)
```

and used as input for the model.

In the literature it is often recommended to scale only the training data with the consequence that the test data must already be scaled in the way mentioned before.

4 Regression

In this chapter, we use a case from the real estate market. The Boston Housing data was collected in 1978 and each of the 506 entries represent aggregated data for homes from various suburbs in Boston, Massachusetts.

There are 14 attributes in each case of the dataset (BostonHousing.csv). They are:

CRIM	per capita crime rate by town
ZN	proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS	proportion of non-retail business acres per town.
CHAS	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
NOX	nitric oxides concentration (parts per 10 million)
RM	average number of rooms per dwelling
AGE	proportion of owner-occupied units built prior to 1940
DIS	weighted distances to five Boston employment centres
RAD	index of accessibility to radial highways
TAX	full-value property-tax rate per \$10,000
PTRATIO	pupil-teacher ratio by town
B	$1000(B_k - 0.63)^2$ where B_k is the proportion of blacks by town
LSTAT	% lower status of the population
MEDV	Median value of owner-occupied homes in \$1000

The aim is to create a regression model which best predicts MEDV.

First, we read and examine the data

```
>>> import os
>>> import pandas as pd
>>> os.chdir("D:/Path")
>>> data_ori = pd.read_csv('BostonHousing.csv')
>>> print(data_ori.shape)
(506, 14)
>>> print(list(data_ori))
['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax',
 'ptratio', 'b', 'lstat', 'medv']
```

```
>>> print(data_ori.describe())
```

	crim	zn	indus	chas	nox	rm	age
count	506.0000	506.0000	506.0000	506.0000	506.0000	506.0000	506.0000
mean	3.6135	11.3636	11.1368	0.0692	0.5547	6.2846	68.5749
std	8.6015	23.3225	6.8604	0.2540	0.1159	0.7026	28.1489
min	0.0063	0.0000	0.4600	0.0000	0.3850	3.5610	2.9000
25%	0.0820	0.0000	5.1900	0.0000	0.4490	5.8855	45.0250
50%	0.2565	0.0000	9.6900	0.0000	0.5380	6.2085	77.5000
75%	3.6771	12.5000	18.1000	0.0000	0.6240	6.6235	94.0750
max	88.9762	100.0000	27.7400	1.0000	0.8710	8.7800	100.0000

	dis	rad	tax	ptratio	b	lstat	medv
count	506.0000	506.0000	506.0000	506.0000	506.0000	506.0000	506.0000
mean	3.7950	9.5494	408.2372	18.4555	356.6740	12.6531	22.5328
std	2.1057	8.7073	168.5371	2.1649	91.2949	7.1411	9.1971
min	1.1296	1.0000	187.0000	12.6000	0.3200	1.7300	5.0000
25%	2.1002	4.0000	279.0000	17.4000	375.3775	6.9500	17.0250
50%	3.2074	5.0000	330.0000	19.0500	391.4400	11.3600	21.2000
75%	5.1884	24.0000	666.0000	20.2000	396.2250	16.9550	25.0000
max	12.1265	24.0000	711.0000	22.0000	396.9000	37.9700	50.0000

Next, we normalize the data

```
>>> data=(data_ori-data_ori.min())/(data_ori.max()-data_ori.min())
```

```
>>> print(data.head(6))
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax
0	0.0000	0.18	0.0678	0.0	0.3148	0.5775	0.6416	0.2692	0.0000	0.2080
	0.2872	1.0000	0.0897	0.4222						
1	0.0002	0.00	0.2423	0.0	0.1728	0.5480	0.7827	0.3490	0.0435	0.1050
	0.5532	1.0000	0.2045	0.3689						
2	0.0002	0.00	0.2423	0.0	0.1728	0.6944	0.5994	0.3490	0.0435	0.1050
	0.5532	0.9897	0.0635	0.6600						
3	0.0003	0.00	0.0630	0.0	0.1502	0.6586	0.4418	0.4485	0.0870	0.0668
	0.6489	0.9943	0.0334	0.6311						
4	0.0007	0.00	0.0630	0.0	0.1502	0.6871	0.5283	0.4485	0.0870	0.0668
	0.6489	1.0000	0.0993	0.6933						
5	0.0003	0.00	0.0630	0.0	0.1502	0.5497	0.5747	0.4485	0.0870	0.0668
	0.6489	0.9930	0.0960	0.5267						

Furthermore, we have to split the data into the input matrix and the target vector. The target variable to forecast is *medv*

```
>>> X = data.drop('medv', axis = 1)
>>> Y = data['medv']
```

Now, we can partition the data via

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                                         test_size = 0.2, random_state=0)
>>> print(Y_train.shape)
(404,)
>>> print(Y_test.shape)
(102,)
```

To evaluate the prediction quality of the models, we will use the R^2 measure. As a benchmark, we initially calculate the mean value and the residual sum of squares of the target variable

```

>>> Y_train_mean = Y_train.mean()
>>> print("Y_train_mean =", Y_train_mean)
Y_train_mean = 0.3913751375137516
>>> Y_train_meandev = sum((Y_train-Y_train_mean)**2)
>>> print("Y_train_meandev =", Y_train_meandev)
Y_train_meandev = 16.986697763109646
>>> Y_test_meandev = sum((Y_test-Y_train_mean)**2)
>>> print("Y_test_meandev =", Y_test_meandev)
Y_test_meandev = 4.109331545079699

```

To compare the results of the models, we create a dataframe

```

>>> report = pd.DataFrame(columns=['Model', 'R2.Train', 'R2.Test'])

```

4.1 Linear Regression

At first, a linear regression using the class *LinearRegression* from the module *sklearn.linear_model* is performed

```

>>> from sklearn.linear_model import LinearRegression
>>> lm = LinearRegression()
>>> lm.fit(X_train, Y_train)

```

Now, we apply the model to the training set

```

>>> Y_train_pred = lm.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.7730135569264233

```

To proof result this, we apply the model to the test set

```

>>> Y_test_pred = lm.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.5899971826290296

```

Finally, we add the result to the report dataframe

```

>>> report.loc[len(report)] = ['OLS Regression', r2, pseudor2]

```

4.2 Ridge Regression

Ridge Regression is implemented in the same module with the class *Ridge*. The lambda parameter is named *alpha* here. To fit a ridge regression model we use

```
>>> from sklearn.linear_model import Ridge
>>> ridgereg = Ridge(alpha=2)
>>> ridgereg.fit(X_train, Y_train)
>>> Y_train_pred = ridgereg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.7623265976032129
>>> Y_test_pred = ridgereg.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.5518604924568339
```

To find the most generalizable model, we need to detect the optimal alpha (lambda) based on the test data. To do this, we write our own procedure.

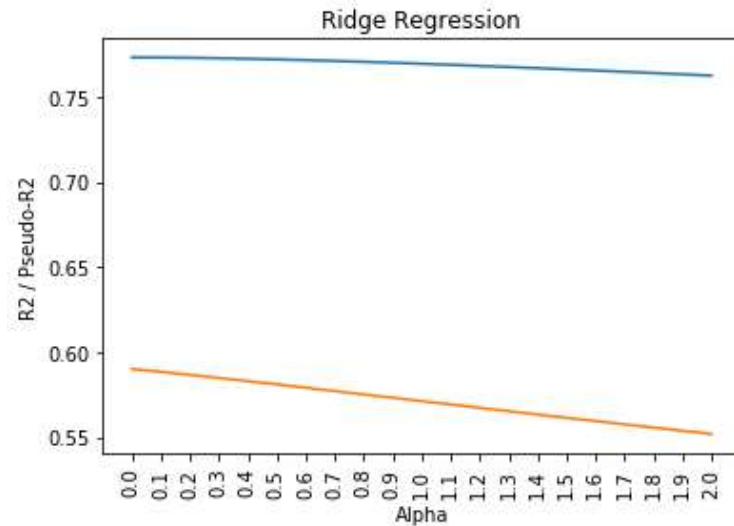
At first, we create a sequence of alphas to try and then use them to train and predict with the corresponding models in a loop

```
>>> r2s = np.zeros((3,21), float)
>>> alphas = np.linspace(0, 2, 21)
>>> for k in range(0, 21):
    ridgereg = Ridge(alpha=alphas[k])
    ridgereg.fit(X_train, Y_train)
    Y_train_pred = ridgereg.predict(X_train)
    Y_train_dev = sum((Y_train-Y_train_pred)**2)
    r2 = 1 - Y_train_dev/Y_train_meandev
    r2s[1,k] = r2
    Y_test_pred = ridgereg.predict(X_test)
    Y_test_dev = sum((Y_test-Y_test_pred)**2)
    pseudor2 = 1 - Y_test_dev/Y_test_meandev
    r2s[2,k] = pseudor2
    r2s[0,k] = alphas[k]
```

To visualize the relations between the lambdas and the R2s/Pseudo-R2s, we plot the results

```
>>> plt.plot(alphas, r2s[1,:])
>>> plt.plot(alphas, r2s[2,:])
```

```
>>> plt.xticks(alphas, rotation=90)
>>> plt.xlabel('Alpha')
>>> plt.ylabel('R2 / Pseudo-R2')
>>> plt.title('Ridge Regression')
>>> plt.show()
```



The results can be visualized in table form via

```
>>> from tabulate import tabulate
>>> headers = ["Lambda", "R2", "Pseudo-R2"]
>>> table = tabulate(r2s.transpose(), headers, tablefmt="plain",
                    floatfmt=".5f")

>>> print("\n",table)

  Alpha      R2      Pseudo-R2
0.00000  0.77301      0.59000
0.10000  0.77295      0.58833
0.20000  0.77279      0.58657
0.30000  0.77254      0.58474
0.40000  0.77222      0.58286
0.50000  0.77184      0.58094
0.60000  0.77141      0.57900
0.70000  0.77093      0.57704
0.80000  0.77042      0.57507
0.90000  0.76986      0.57310
1.00000  0.76928      0.57113
1.10000  0.76867      0.56916
1.20000  0.76804      0.56719
1.30000  0.76738      0.56524
1.40000  0.76670      0.56329
```

1.50000	0.76601	0.56135
1.60000	0.76530	0.55943
1.70000	0.76458	0.55752
1.80000	0.76384	0.55562
1.90000	0.76309	0.55373
2.00000	0.76233	0.55186

Now, we extract the maximum of Pseudo-R2 and the corresponding alpha:

```
>>> maxi = np.array(np.where(r2s==r2s[2].max()))
>>> table = tabulate(r2s[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".5f")
>>> print("\n",table)
      Alpha      R2      Pseudo-R2
0.00000  0.77301      0.59000
```

Finally, we apply it:

```
>>> ridgereg = Ridge(alpha=r2s[0,maxi[1,:]])
>>> ridgereg.fit(X_train, Y_train)
>>> Y_train_pred = ridgereg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.7730135569264233
>>> Y_test_pred = ridgereg.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.5899971826290296
>>> report.loc[len(report)] = ['Ridge Regression', r2, pseudor2]
```

In this case, regularization has no effect.

4.3 Support Vector Regression

To perform a Support Vector Regression SVR class from the module `sklearn.svm` can be used (see 3.7). In this case, we start with a linear kernel, `epsilon=0.1`, and `cost=1`

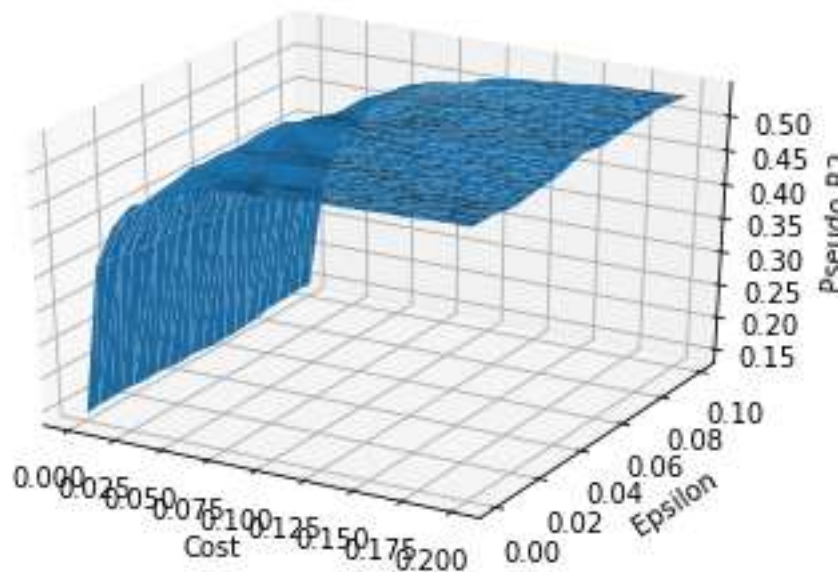
```
>>> from sklearn.svm import SVR
>>> LinSVRreg = SVR(kernel='linear', C=1.0, epsilon=0.1)
>>> LinSVRreg.fit(X_train, Y_train)
```

Now, we apply the model and calculate the R^2 -measures


```
>>> print("\n",table)
      Cost      Epsilon      R2      Pseudo-R2
0.200      0.085  0.762      0.540
```

We can create a surface plot via

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> x = r2s[0,:]
>>> y = r2s[1,:]
>>> z = r2s [3,:]
>>> ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)
>>> ax.set_xlabel('Cost')
>>> ax.set_ylabel('Epsilon')
>>> ax.set_zlabel('Pseudo-R2')
>>> plt.show()
```



Now, we can use the best parameters to fit the linear svr model

```
>>> LinSVRreg = SVR(kernel='linear', C=r2s[0,maxi[1,:]],
                    epsilon=r2s[1,maxi[1,:]])
>>> LinSVRreg.fit(X_train, Y_train)
>>> Y_train_pred = LinSVRreg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
R2 = 0.762195835499534
>>> print("R2 =", r2)
```



```

>>> Y_test_pred = LinSVRreg.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.5397311929356355
>>> report.loc[len(report)] = ['Linear SVR', r2, pseudor2]

```

Until now, we used only linear estimators in this chapter. The results indicate that the relationships might be not linear. To proof this, we apply the radial kernel

```

>>> RbfSVRreg = SVR(kernel='rbf', C=1.0, epsilon=0.1)
>>> RbfSVRreg.fit(X_train, Y_train)
>>> Y_train_pred = RbfSVRreg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.7900762681663058
>>> Y_test_pred = RbfSVRreg.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.5586486953702154

```

The result is similar to that of the linear kernel. To optimize the parameters, we use the procedure as applied in the linear case but now we have to search for three parameters: cost, epsilon, and gamma

```

>>> n = 21
>>> r2s = np.zeros((5,n*n*n), float)
>>> costs = np.linspace(0, 3, n)
>>> costs[0] = 0.001
>>> epsilons = np.linspace(0, 0.1, n)
>>> gammas = np.linspace(0, 4.0, n)
>>> gammas[0] = 0.1
>>> row = 0
>>> for k in range(0, n):
    for l in range(0, n):
        for m in range(0, n):
            RbfSVRreg = SVR(kernel='rbf', C=costs[k],
                               epsilon=epsilons[l], gamma=gammas[m])
            RbfSVRreg.fit(X_train, Y_train)
            Y_train_pred = RbfSVRreg.predict(X_train)

```

```

Y_train_dev = sum((Y_train-Y_train_pred)**2)
r2 = 1 - Y_train_dev/Y_train_meandev
r2s[2,row] = r2
Y_test_pred = RbfSVRreg.predict(X_test)
Y_test_dev = sum((Y_test-Y_test_pred)**2)
pseudor2 = 1 - Y_test_dev/Y_test_meandev
r2s[3,row] = pseudor2
r2s[0,row] = costs[k]
r2s[1,row] = epsilons[l]
r2s[2,row] = gammas[m]
row = row + 1

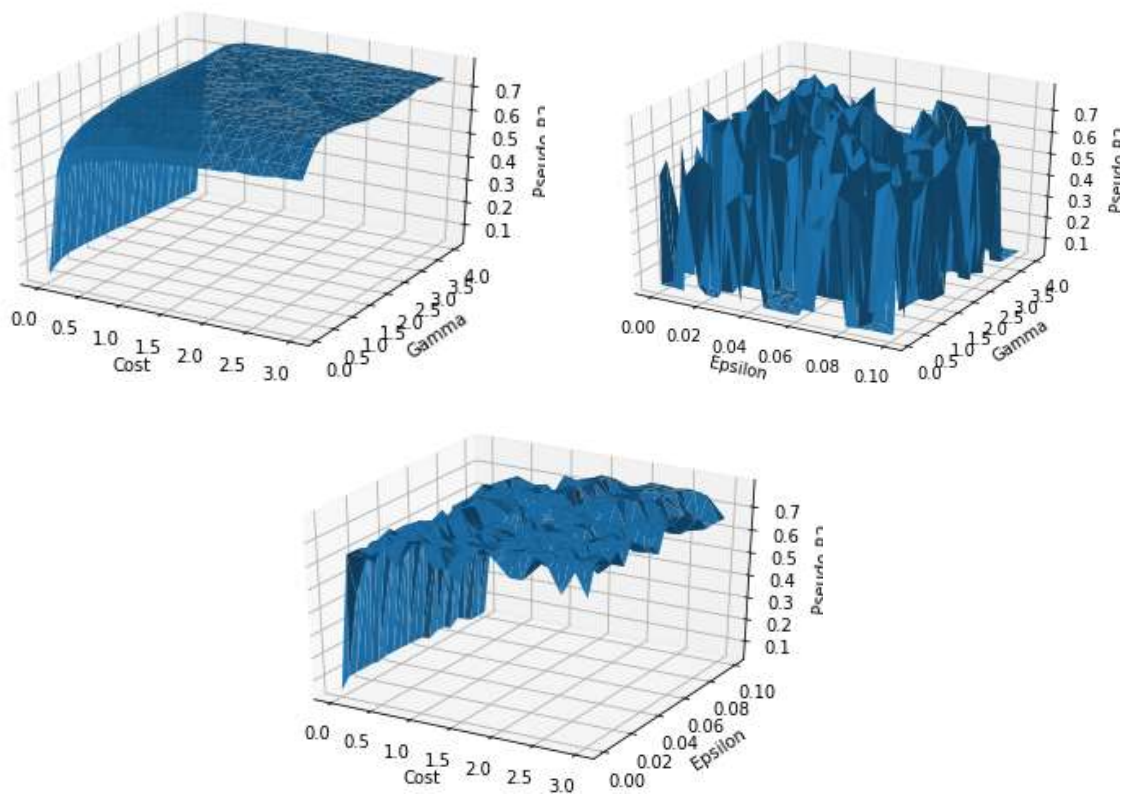
>>> maxi = np.array(np.where(r2s==r2s[4].max()))
>>> headers = ["Cost", "Epsilon", "Gamma", "R2", "Pseudo-R2"]
>>> table = tabulate(r2s[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".3f")

>>> print("\n",table)

```

Cost	Epsilon	Gamma	R2	Pseudo-R2
2.250	0.065	0.800	0.941	0.799

The surface graphs show the relations between the parameters



Using the parameter values of the first row in the matrix *maxi*

```
RbfSVRreg = SVR(kernel='rbf', C=r2s[0,maxi[1,0]],
                 epsilon=r2s[1,maxi[1,0]], gamma=r2s[2,maxi[1,0]])
```

we obtain $R^2 = 0.9408491605669842$ and Pseudo- $R^2 = 0.7994523905927886$.

4.4 Neural Networks

The application of neural networks is similar to the description in section 3.6. The class name is *MLPRegressor* here

```
>>> from sklearn.neural_network import MLPRegressor
>>> NNetRreg = MLPRegressor(solver='lbfgs', hidden_layer_sizes=(10,),
                           random_state=0)
>>> NNetRreg.fit(X_train, Y_train)
>>> Y_train_pred = NNetRreg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.8903613573604956
>>> Y_test_pred = NNetRreg.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.7485264410158461
```

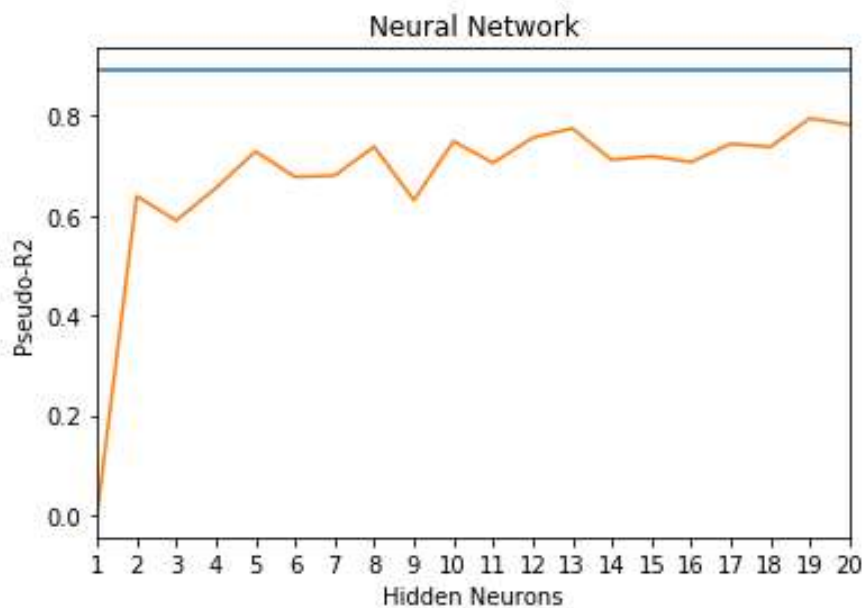
The result shows the typical behavior of neural networks to overfit. The training data is fitted nearly perfect, but the result for test data is very poor. We vary the number of hidden neurons to improve the result

```
>>> r2s = np.zeros((3,20), float)
>>> for k in range(0, 20):
    NNetRreg = MLPRegressor(solver='lbfgs',
                           hidden_layer_sizes=(k+1,), random_state=0)
    NNetRreg.fit(X_train, Y_train)
    Y_train_pred = NNetRreg.predict(X_train)
    r2 = 1 - Y_train_dev/Y_train_meandev
    r2s[1,k] = r2
    Y_test_pred = NNetRreg.predict(X_test)
    Y_test_dev = sum((Y_test-Y_test_pred)**2)
    pseudor2 = 1 - Y_test_dev/Y_test_meandev
    r2s[2,k] = pseudor2
```

```

r2s[0,k] = k+1
>>> plt.plot(r2s[0,:], r2s[1,:])
>>> plt.plot(r2s[0,:], r2s[2,:])
>>> plt.xlim(1,20)
>>> plt.xticks(r2s[0,:])
>>> plt.xlabel('Hidden Neurons')
>>> plt.ylabel('Pseudo-R2')
>>> plt.title('Neural Network')
>>> plt.show()

```



It can be seen, that the number of hidden neurons has a strong impact on the Pseudo- R^2 .

```

>>> maxi = np.array(np.where(r2s==r2s[2].max()))
>>> table = tabulate(r2s[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".5f")

>>> print("\n",table)
    Hidden Neurons      R2      Pseudo-R2
         19.00000    0.89036      0.79484

>>> NNetRreg = MLPRegressor(solver='lbfgs',
                          hidden_layer_sizes=(int(r2s[0,maxi[1,:]]),), random_state=0)
>>> NNetRreg.fit(X_train, Y_train)
>>> Y_train_pred = NNetRreg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.9474036982902446
>>> Y_test_pred = NNetRreg.predict(X_test)

```

```
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.7948372842833415
```

Using the optimal number of hidden neurons, we got a similar result compared to the radial SVR.

To handle overfitting, *MLPRegressor* provides a regularization mechanism analogous to Ridge Regression which can be used via the parameter *alpha*. Usually the weights are updated during the learning process via

$$\omega_i(t+1) = \omega_i - \eta \frac{\partial E}{\partial \omega_i}$$

where η is the learning rate. The parameter α (λ) penalizes the weight changes in the following way

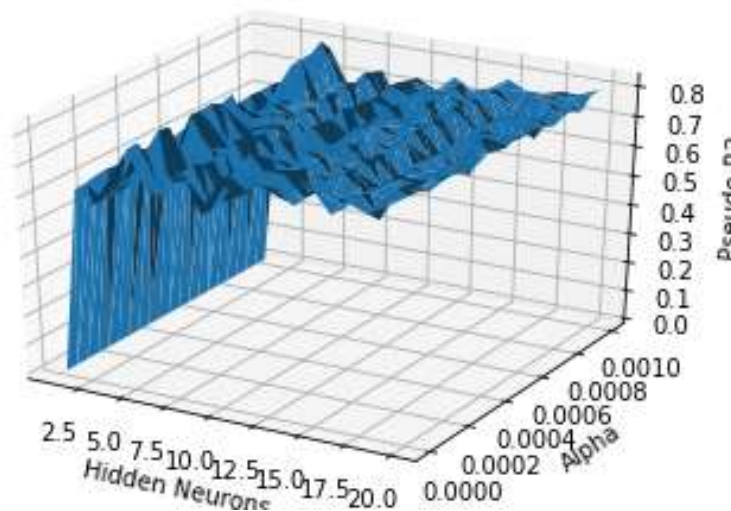
$$\omega_i(t+1) = \omega_i - \eta \frac{\partial E}{\partial \omega_i} - \lambda \eta \omega_i$$

We repeat the model creation varying the hidden neurons and alpha in a grid search

```
>>> alphas = np.linspace(0, 0.001, 21)
>>> r2s = np.zeros((4,20*21), float)
>>> row = 0
>>> for k in range(0, 20):
    for l in range(0, 21):
        NNetRreg = MLPRegressor(solver='lbfgs',
                                hidden_layer_sizes=(k+1,),
                                alpha=alphas[l],
                                random_state=0)
        NNetRreg.fit(X_train, Y_train)
        Y_train_pred = NNetRreg.predict(X_train)
        r2 = 1 - Y_train_dev/Y_train_meandev
        r2s[2,row] = r2
        Y_test_pred = NNetRreg.predict(X_test)
        Y_test_dev = sum((Y_test-Y_test_pred)**2)
        pseudor2 = 1 - Y_test_dev/Y_test_meandev
        r2s[3,row] = pseudor2
        r2s[0,row] = k+1
        r2s[1,row] = alphas[l]
        row = row + 1
>>> from tabulate import tabulate
```

```
>>> headers = ["Hidden Neurons", "Alpha", "R2", "Pseudo-R2"]
>>> table = tabulate(r2s[:,maxi[1,:]].transpose(), headers,
    tablefmt="plain", floatfmt=".5f")
>>> print("\n",table)
```

Hidden Neurons	Alpha	R2	Pseudo-R2
5.00000	0.00095	0.94740	0.82450



The result shows that in this case the regularization has a small impact. The result could slightly be improved with $R^2 = 0.9180563262849575$ and Pseudo- $R^2 = 0.8245023064137195$

4.5 Random Forests

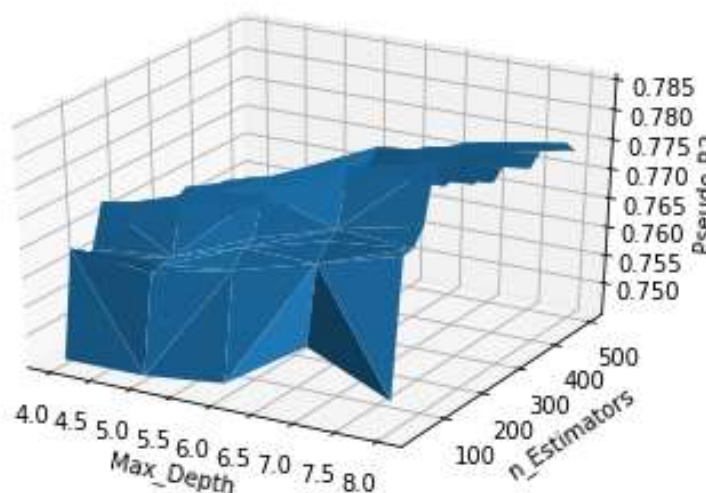
The application of the Random Forest method in the regression case is equal to the application in the classification case. Here the regression functionality is implemented in class *RandomForestRegressor*

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> RForreg = RandomForestRegressor(n_estimators=500, random_state=0)
>>> RForreg.fit(X_train, Y_train)
>>> Y_train_pred = RForreg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.9837797967146643
>>> Y_test_pred = RForreg.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.7775910723784181
```

We vary max_depth and n_estimators

```
>>> mdepth = np.linspace(4, 8, 5)
>>> ntrees = (np.arange(20)+1)*25
>>> r2s = np.zeros((4, 5*20), float)
>>> row = 0
>>> for k in range(0, 5):
    for l in range(0, 20):
        RForreg = RandomForestRegressor(max_depth=mdepth[k],
                                         n_estimators=ntrees[l], random_state=0)
        RForreg.fit(X_train, Y_train)
        Y_train_dev = sum((Y_train-Y_train_pred)**2)
        r2 = 1 - Y_train_dev/Y_train_meandev
        r2s[2,row] = r2
        Y_test_pred = RForreg.predict(X_test)
        Y_test_dev = sum((Y_test-Y_test_pred)**2)
        pseudor2 = 1 - Y_test_dev/Y_test_meandev
        r2s[3,row] = pseudor2
        r2s[0,row] = mdepth[k]
        r2s[1,row] = ntrees[l]
        row = row + 1
>>> from tabulate import tabulate
>>> headers = ["Max_Depth", "n_Estimators", "R2", "Pseudo-R2"]
>>> table = tabulate(r2s[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".5f")
>>> print("\n",table)
```

Max_Depth	n_Estimators	R2	Pseudo-R2
8.00000	125.00000	0.98378	0.78507



4.6 Gradient Boosting

The gradient boosting algorithm for regression is implemented in *GradientBoostingRegressor*. It is very similar to the *GradientBoostingClassifier*.

We apply it with the following code

```
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> GBoostreg = GradientBoostingRegressor(random_state=0)
>>> GBoostreg.fit(X_train, Y_train)
>>> Y_train_pred = GBoostreg.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.9819844794245283
>>> Y_test_pred = GBoostreg.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.7771416836956806
```

To tune the parameters, we vary *max_depth* and *learning_rate*

```
>>> r2s = np.zeros((4,21*5), float)
>>> lr = np.linspace(0, 0.4, 21)
>>> lr[0] = 0.01
>>> row = 0
>>> for k in range(0, 5):
    for l in range(0, 21):
        GBoostreg = GradientBoostingRegressor(random_state=0,
                                                max_depth=k+1, learning_rate=lr[l])
        GBoostreg.fit(X_train, Y_train)
        Y_train_dev = sum((Y_train-Y_train_pred)**2)
        r2 = 1 - Y_train_dev/Y_train_meandev
        r2s[2,row] = r2
        Y_test_pred = GBoostreg.predict(X_test)
        Y_test_dev = sum((Y_test-Y_test_pred)**2)
        pseudor2 = 1 - Y_test_dev/Y_test_meandev
        r2s[3,row] = pseudor2
        r2s[0,row] = k+1
        r2s[1,row] = lr[l]
        row = row + 1
>>> from tabulate import tabulate
```

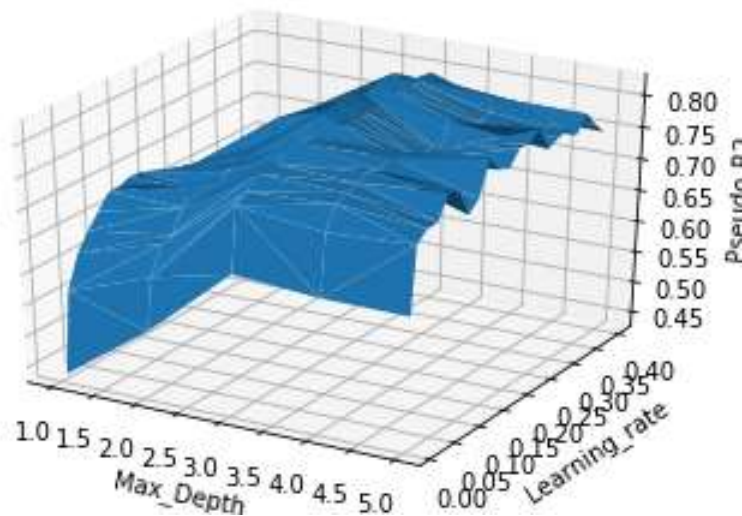


```

>>> headers = ["Max_Depth", "Learning_rate", "R2", "Pseudo-R2"]
>>> table = tabulate(r2s[:,maxi[1,:]].transpose(), headers,
                    tablefmt="plain", floatfmt=".5f")

>>> print("\n",table)
    Max_Depth    Learning_rate      R2    Pseudo-R2
      3.00000      0.32000  0.98198    0.82502
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> x = r2s[0,:]
>>> y = r2s[1,:]
>>> z = r2s[3,:]
>>> ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)
>>> ax.set_xlabel('Max_Depth')
>>> ax.set_ylabel('Learning_rate')
>>> ax.set_zlabel('Pseudo-R2')
>>> plt.show()

```



Finally, we get:

```

>>> print(report)

```

	Model	R2.Train	R2.Test
0	OLS Regression	0.7730	0.5900
1	Ridge Regression	0.7730	0.5900
2	Linear SVR	0.7622	0.5397
3	Radial SVR	0.9408	0.7995
4	Neural Network	0.9181	0.8245
5	Random Forest	0.9767	0.7851
6	Gradient Boosting	0.9974	0.8250


```

>>> CV_rrmodel.fit(X_train, Y_train)
>>> print(CV_rrmodel.best_params_)
{'alpha': 0.5}
>>> ridgeregCV = ridgeregCV.set_params(**CV_rrmodel.best_params_)
>>> ridgeregCV.fit(X_train, Y_train)
>>> Y_train_pred = ridgeregCV.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.7718440443900478
>>> Y_test_pred = ridgeregCV.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
>>> report.loc[len(report)] = ['Ridge RegressionCV', r2, pseudor2]
Pseudo-R2 = 0.580940441291366

>>> # Support Vector Regression
>>> from sklearn.svm import SVR
>>> RbfSVRregCV = SVR()
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {
>>>     'kernel': ["linear", "rbf"],
>>>     'C': [1, 3, 5, 8, 10],
>>>     'epsilon': [0.0, 0.025, 0.05, 0.075, 0.1],
>>>     'gamma' : [0., 1., 2., 3., 4.]
>>> }
>>> CV_svrmodel = GridSearchCV(estimator=RbfSVRregCV,
                              param_grid=param_grid, cv=10)
>>> CV_svrmodel.fit(X_train, Y_train)
>>> print(CV_svrmodel.best_params_)
{'C': 3, 'epsilon': 0.025, 'gamma': 1.0, 'kernel': 'rbf'}
>>> RbfSVRregCV = RbfSVRregCV.set_params(**CV_svrmodel.best_params_)
>>> RbfSVRregCV.fit(X_train, Y_train)
>>> Y_train_pred = RbfSVRregCV.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.9642955633242669

```

```

>>> Y_test_pred = RbfSVRregCV.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
>>> report.loc[len(report)] = ['Support Vector RegressionCV', r2,
    pseudor2]
Pseudo-R2 = 0.7915880573721309

>>> # Neural Network
>>> from sklearn.neural_network import MLPRegressor
>>> NNetRregCV = MLPRegressor(solver='lbfgs', random_state=0)
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {
>>>     'learning_rate': ["constant", "invscaling", "adaptive"],
>>>     'hidden_layer_sizes': [(5,), (8,), (10,), (13,)],
>>>     'alpha': [0.0, 0.0025, 0.005, 0.0075, 0.01, 0.1],
>>>     'activation': ["logistic", "relu", "tanh"]
>>> }
>>> CV_nnmodel = GridSearchCV(estimator=NNetRregCV,
    param_grid=param_grid, cv=10)
>>> CV_nnmodel.fit(X_train, Y_train)
>>> print(CV_nnmodel.best_params_)
{'activation': 'tanh', 'alpha': 0.0075, 'hidden_layer_sizes': (10,),
    'learning_rate': 'constant'}
>>> NNetRregCV = NNetRregCV.set_params(**CV_nnmodel.best_params_)
>>> NNetRregCV.fit(X_train, Y_train)
>>> Y_train_pred = NNetRregCV.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.9275042728823828
>>> Y_test_pred = NNetRregCV.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
>>> report.loc[len(report)] = ['Neural NetworkCV', r2, pseudor2]
Pseudo-R2 = 0.7826520749703318

```

```
>>> # Random Forest
```

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> RForregCV = RandomForestRegressor(random_state=0)
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {
>>>     'max_depth': [ 4.,  5.,  6.,  7.,  8.],
>>>     'n_estimators': [ 10,  50, 100, 150, 200]
>>> }
>>> CV_rfmodel = GridSearchCV(estimator=RForregCV,
                             param_grid=param_grid, cv=10)
>>> CV_rfmodel.fit(X_train, Y_train)
>>> print(CV_rfmodel.best_params_)
{'max_depth': 8.0, 'n_estimators': 150}
>>> RForregCV = RForregCV.set_params(**CV_rfmodel.best_params_)
>>> RForregCV.fit(X_train, Y_train)
>>> Y_train_pred = RForregCV.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.9771594759092019
>>> Y_test_pred = RForregCV.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
>>> report.loc[len(report)] = ['Random ForestCV', r2, pseudor2]
Pseudo-R2 = 0.7804699823740516
```

```
>>> # Gradient Boosting
```

[illegible]

```

>>> CV_gbmodel.fit(X_train, Y_train)
>>> print(CV_gbmodel.best_params_)
{'learning_rate': 0.2, 'max_depth': 5.0, 'n_estimators': 150,
  'subsample': 0.9}

>>> GBoostregCV = GBoostregCV.set_params(**CV_gbmodel.best_params_)
>>> GBoostregCV.fit(X_train, Y_train)
>>> Y_train_pred = GBoostregCV.predict(X_train)
>>> Y_train_dev = sum((Y_train-Y_train_pred)**2)
>>> r2 = 1 - Y_train_dev/Y_train_meandev
>>> print("R2 =", r2)
R2 = 0.9999895659986467

>>> Y_test_pred = GBoostregCV.predict(X_test)
>>> Y_test_dev = sum((Y_test-Y_test_pred)**2)
>>> pseudor2 = 1 - Y_test_dev/Y_test_meandev
>>> print("Pseudo-R2 =", pseudor2)
Pseudo-R2 = 0.7587944836023944

```

Finally, we print the comparing report

```

>>> print(report)

```

	Model	R2.Train	R2.Test
0	OLS RegressionCV	0.773014	0.589997
1	Ridge RegressionCV	0.771844	0.580940
2	Support Vector RegressionCV	0.964296	0.791588
3	Neural NetworkCV	0.927504	0.782652
4	Random ForestCV	0.977159	0.780470
5	Gradient BoostingCV	0.999990	0.758794

As one can see, the Pseudo-R2 is worse compared to the models using the test sample for hyperparameter optimization. Furthermore, the ranking of the methods differs. Thus, the results here are more objective.

4.8 Applying a Regression Model to New Data

Even in the case of regression, it must be the case that if a model is trained on scaled data, it will only work with scaled data. Thus, scaling of new data instances can be done in the same way as described in section 3.10 if the target variable itself was not scaled too.

Scaling the target variable is necessary, if the model is only able to produce values between 0 and 1, for example a neural network with a corresponding activation function. Furthermore, scaling might be necessary for the learning process. A target variable with a large spread of

values, in turn, may result in large error gradient values causing weight values to change dramatically, making the learning process unstable.

But if the target variable is scaled, the model will always produce scaled results. Thus the resulting value must be unscaled before it can be used as a prediction. This can be done by calling the `inverse_transform()` function. Example:

```
>>> nscaler = preprocessing.MinMaxScaler()  
>>> data = nscaler.fit_transform(data_ori)  
>>> data_ori = nscaler.inverse_transform(data)
```

5 Interpreting Machine Learning Models

5.1 Skater

There exist several libraries for Python to apply methods to interpret complex ML models. Many concentrate exclusively on single methods, e.g. PDPbox for creating PDPs.

A library that combines several methods under one uniform API is Skater. Skater combines and improves several methods to provide a common framework for describing predictive models, regardless of the algorithm used to build them. This library is to be used here.

At first, we have to build a model. We use the Boston Housing dataset from the previous chapter and create a random forest model

```
>>> import os
>>> import pandas as pd
>>> os.chdir("D:/Dropbox/Lehre/Digital Analytics/Excercises/Regression")
>>> data_ori = pd.read_csv('BostonHousing.csv')
>>> data=(data_ori-data_ori.min())/(data_ori.max()-data_ori.min())
>>> train = data.iloc[0:200,:]
>>> test = data.iloc[200:,:]
>>> X_train = train.drop('medv', axis = 1)
>>> Y_train = train['medv']
>>> X_test = test.drop('medv', axis = 1)
>>> Y_test = test['medv']
>>> feature_names = X_train.columns
>>> print(feature_names)
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad',
      'tax', 'ptratio', 'b', 'lstat'], dtype='object')

>>> from sklearn.ensemble import RandomForestRegressor
>>> RForreg = RandomForestRegressor(n_estimators=500, random_state=0)
>>> RForreg.fit(X_train, Y_train)
```

The general workflow within the skater package is to create an interpretation, create a model, and run interpretation algorithms.

An Interpretation consumes a dataset, and optionally some meta data like feature names and row ids. Internally, the Interpretation will generate a DataManager to handle data requests and sampling

```
>>> from skater.core.explanations import Interpretation
>>> interpreter = Interpretation()
```


To begin using the Interpretation to explain models, we need to create a skater model. To create a skater model based on a local function or method, pass in the predict function to an InMemoryModel

```
>>> from skater.model import InMemoryModel
>>> annotated_model = InMemoryModel(RForreg.predict, examples=X_test)
```

The skater model provides several informations

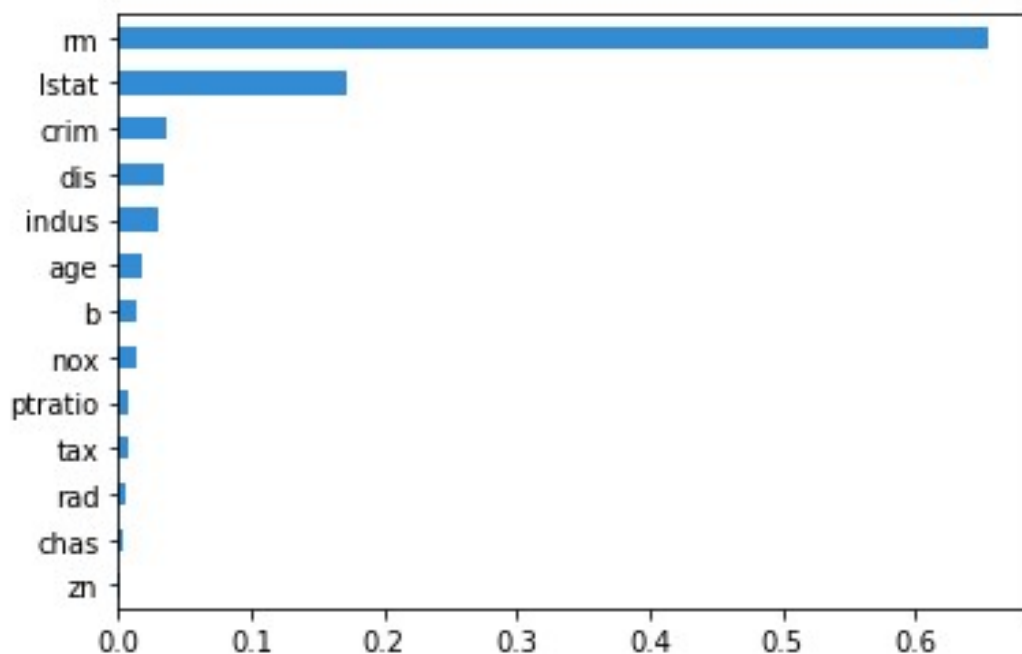
```
>>> print("Number of classes: {}".format(annotated_model.n_classes))
Number of classes: 1
>>> print("Input shape: {}".format(annotated_model.input_shape))
Input shape: (306, 13)
>>> print("Model Type: {}".format(annotated_model.model_type))
Model Type: regressor
>>> print("Output Shape: {}".format(annotated_model.output_shape))
Output Shape: (306,)
>>> print("Output Type: {}".format(annotated_model.output_type))
Output Type: continuous
```

The data must now be loaded and linked to the names of the features

```
>>> interpreter.load_data(X_test, feature_names=feature_names)
```

To calculate and display the **Feature Importance**, we use

```
>>> interpreter.feature_importance.feature_importance(annotated_model)
>>> interpreter.feature_importance.plot_feature_importance(
    annotated_model, ascending=False)
```

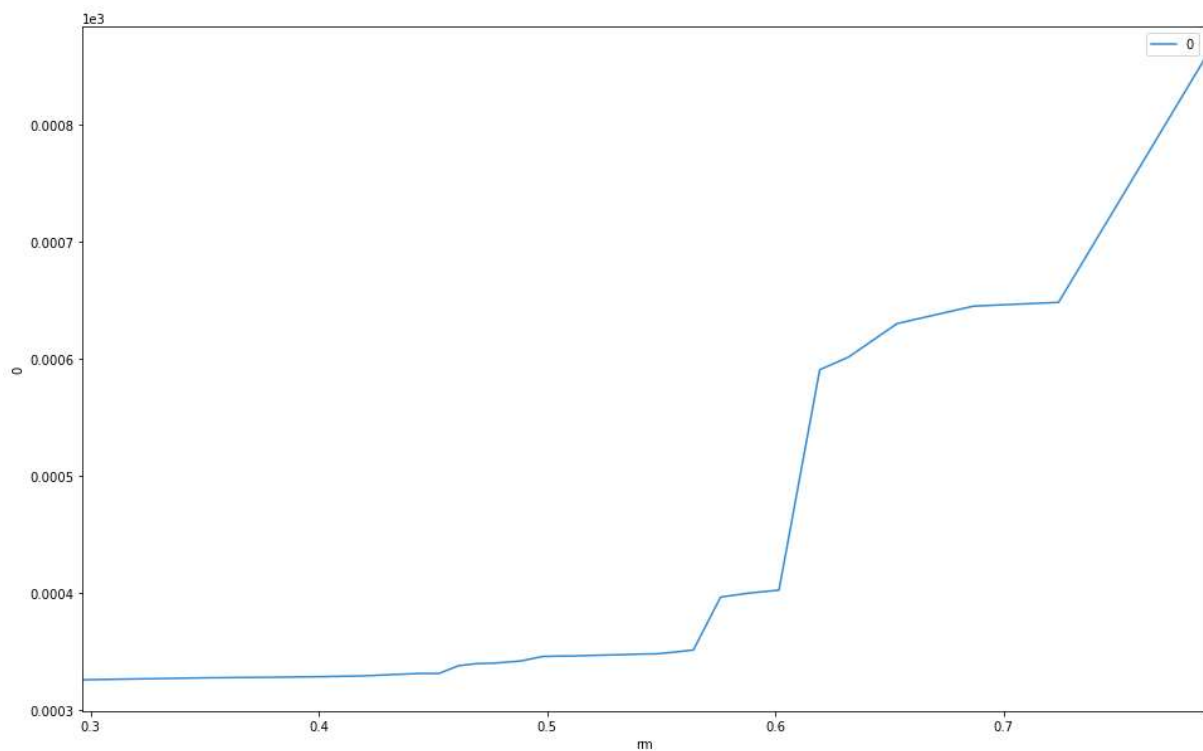
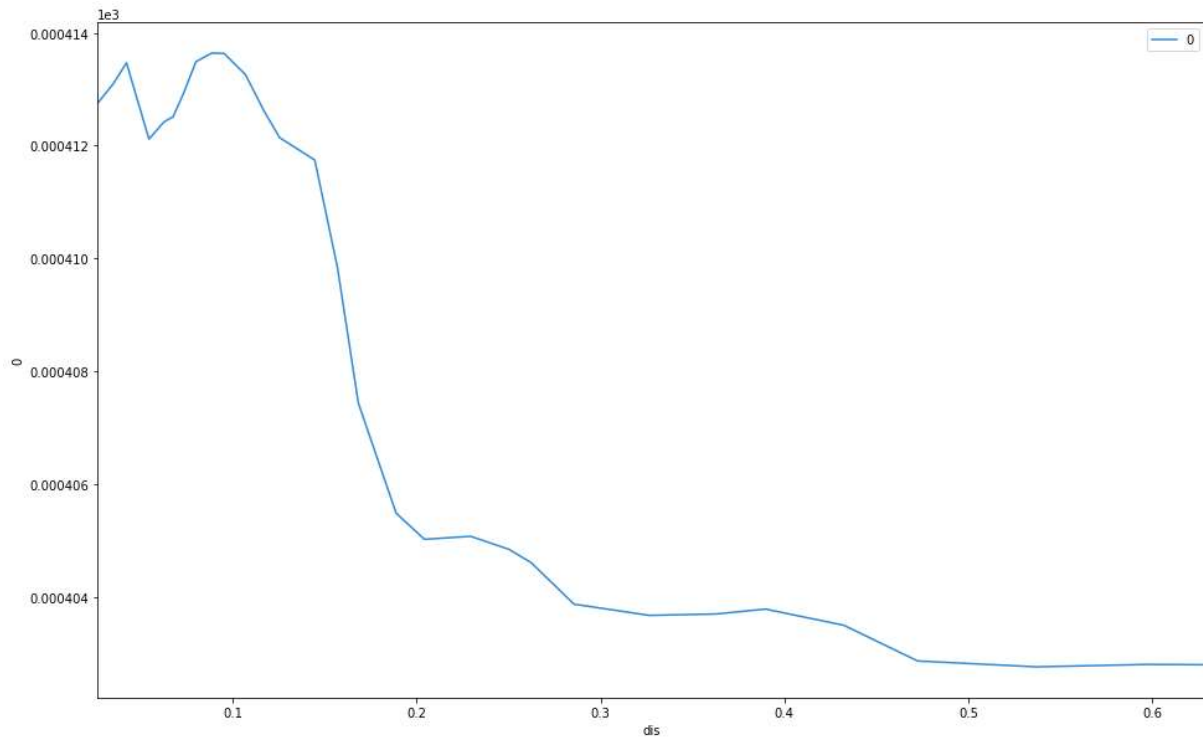


Partial Dependence Plots can be created based on a list of features

```
>>> pdp_features = ['dis', 'rm']
```

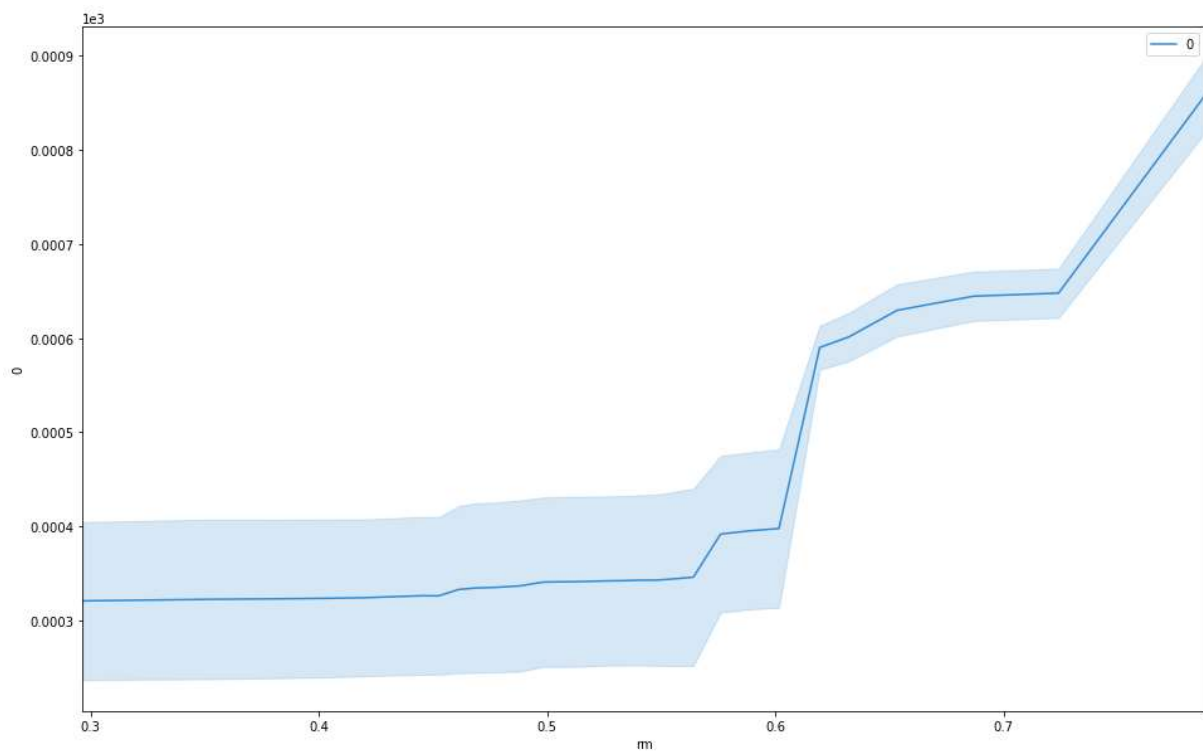
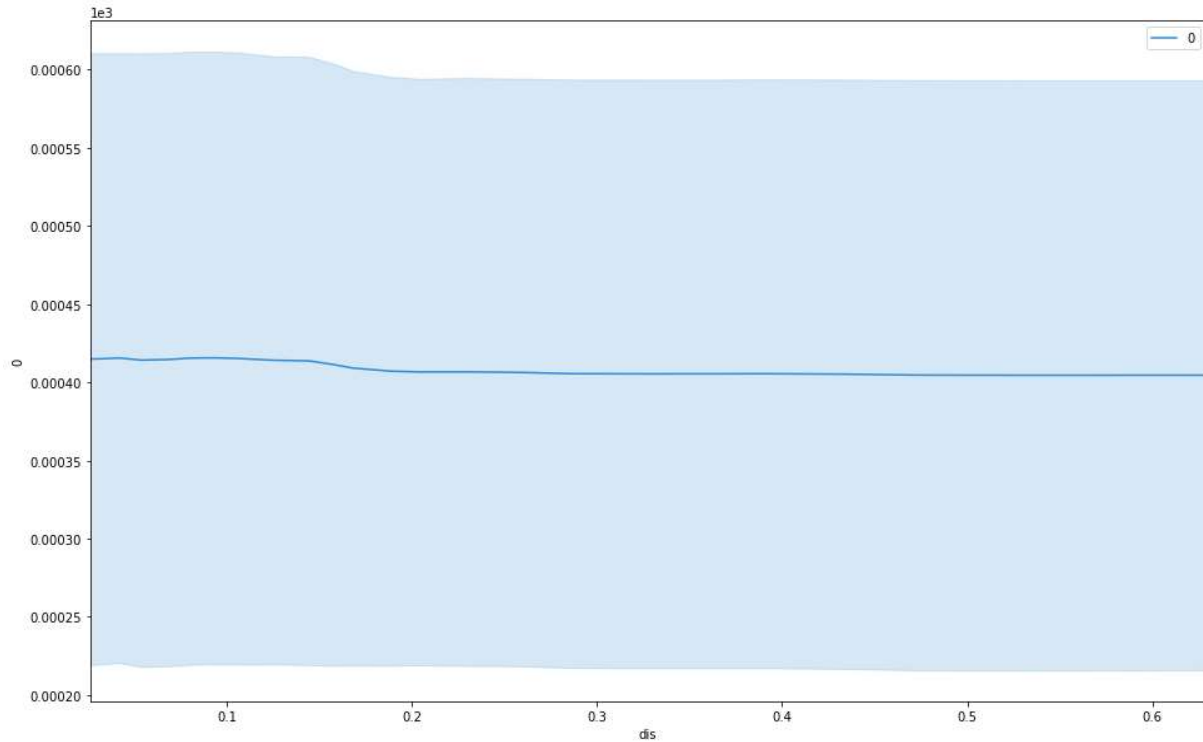
To create 1-way partial dependence plots, we use

```
>>> interpreter.partial_dependence.plot_partial_dependence(  
    pdp_features, annotated_model, grid_resolution=30)
```



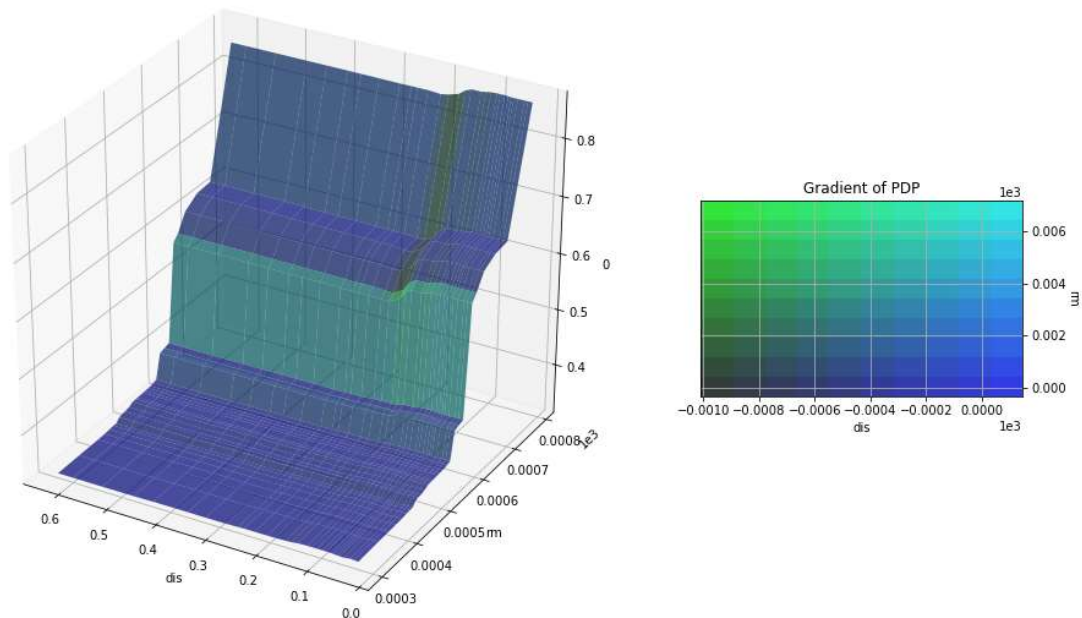
To create 1-way partial dependence plots with variance effect, we use

```
>>> interpreter.partial_dependence.plot_partial_dependence(
    pdp_features, annotated_model, grid_resolution=30,
    with_variance=True)
```



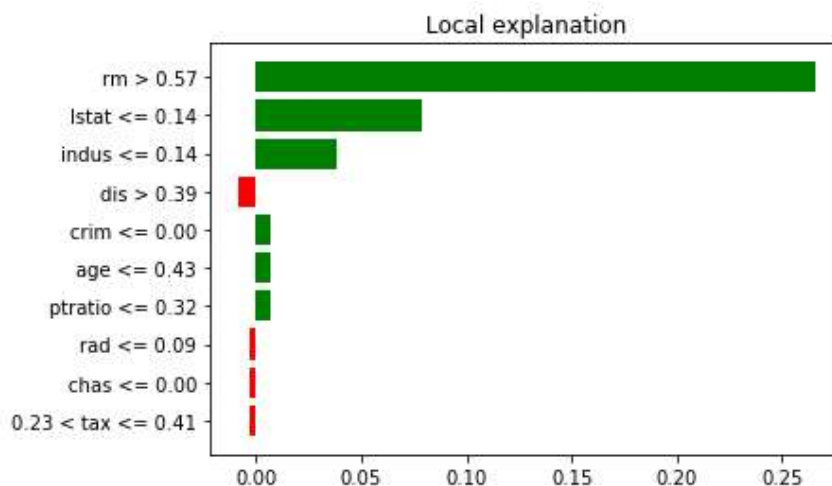
By linking the command, we can create 2-way partial dependence plots

```
>>> interpreter.partial_dependence.interpreter.partial_dependence.  
plot_partial_dependence(pdp_features, annotated_model,  
grid_resolution=30)
```



Skater contains functions to apply **Lime**

```
>>> from skater.core.local_interpretation.lime.lime_tabular import  
LimeTabularExplainer  
  
>>> explainer = LimeTabularExplainer(X_train.values,  
feature_names=feature_names, mode="regression")  
  
>>> explanation = explainer.explain_instance(X_test.iloc[2, :],  
annotated_model)  
  
>>> explanation.as_list()  
>>> explanation.as_pyplot_figure(); # ";" to avoid two plots
```

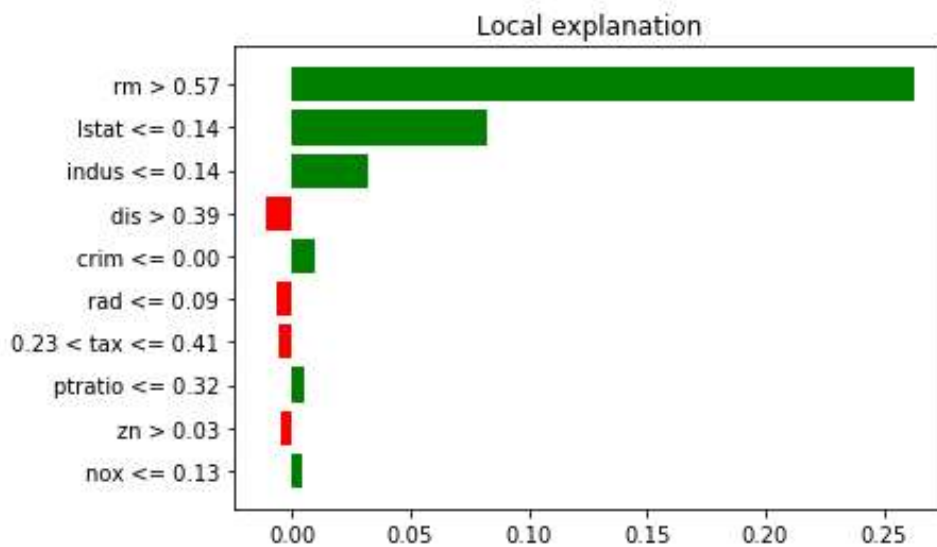


5.2 Lime

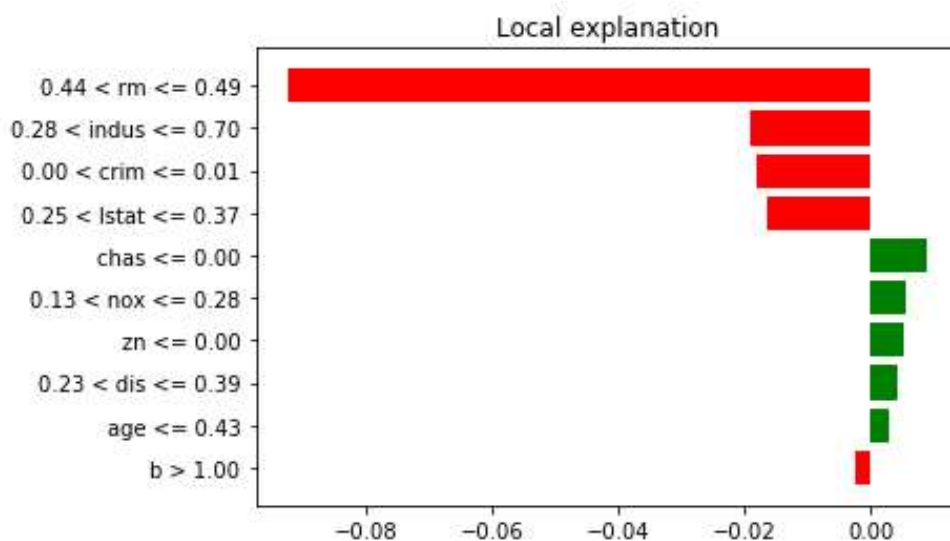
With an installed Lime library, the functions can also be used without a skater

```
>>> from lime.lime_tabular import LimeTabularExplainer
>>> explainer = LimeTabularExplainer(X_train.values,
                                     feature_names = feature_names,
                                     mode='regression', random_state=1)

>>> exp = explainer.explain_instance(X_test.iloc[2, :], RForreg.predict)
>>> exp.as_list()
>>> exp.as_pyplot_figure(); # ";" to avoid two plots
```



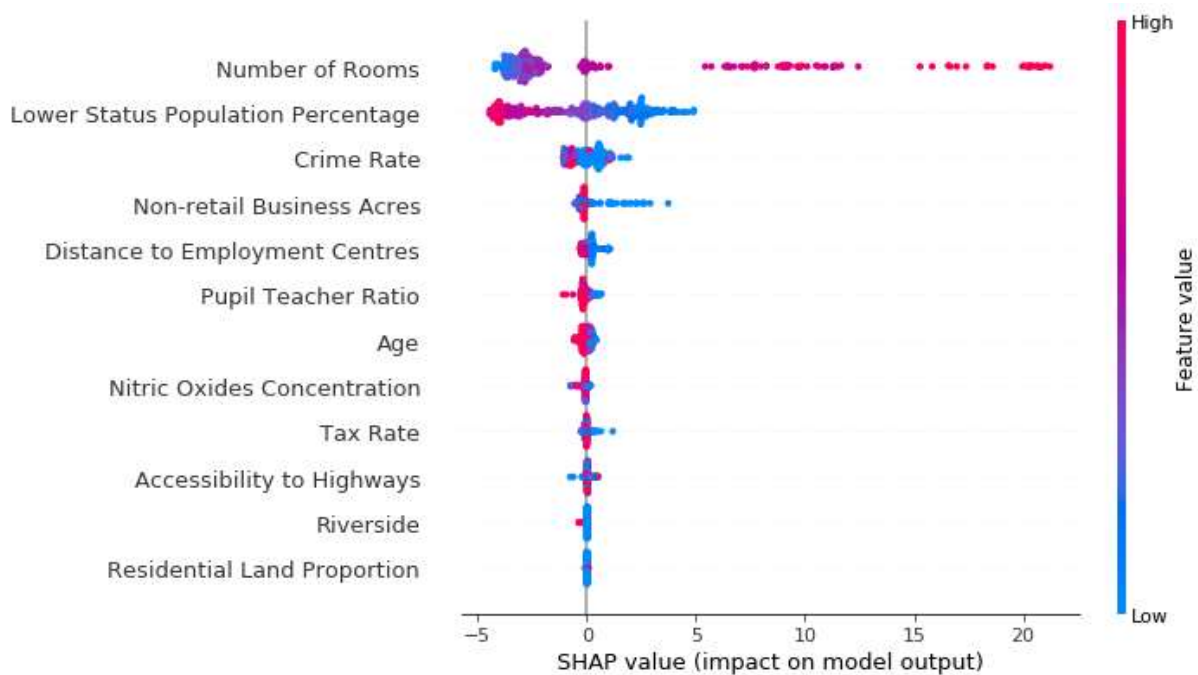
```
>>> exp = explainer.explain_instance(X_test.iloc[5, :], RForreg.predict)
>>> exp.as_list()
>>> exp.as_pyplot_figure(); # ";" to avoid two plots
```



5.3 Shap

The calculation of Shapley Values is implemented in the library Shap

```
>>> import shap
>>> shap_explainer = shap.TreeExplainer(RForreg)
>>> test_shap_vals = shap_explainer.shap_values(X_test)
>>> shap.summary_plot(test_shap_vals, X_test)
```



This summary plot replaces the typical bar chart of feature importance. It tells which features are most important, and also their range of effects over the dataset. The color allows us match how changes in the value of a feature effect the change in the target.

Thus, each dot has three characteristics:

- Horizontal location shows what feature it is depicting
- Color shows whether that feature was high or low for that row of the dataset
- Vertical location shows whether the effect of that value caused a higher or lower prediction.

For example, the higher the number of rooms the higher the price will be.

6 Segmentation

Scikit-learn provides various methods of segmentation. In this chapter, we use the case of a real estate agent who sells various houses. To distinguish between different types of houses, the agent wants to perform a segmentation using all of his house objects. The data is stored in a CSV file.

As a first step, the path to the directory of the file is set and the file is read into a dataframe

```
>>> import os
>>> import numpy as np
>>> import pandas as pd

>>> os.chdir("D:/Path")
>>> data_ori = pd.read_csv('Housing2.csv')
>>> print(data_ori.shape)
(506, 12)
```

Because the features are on a different scale level, the `scale()` function is applied next

```
>>> data=(data_ori-data_ori.min())/(data_ori.max()-data_ori.min())
>>> print(data.describe())
```

	HousePrice	HouseSize	HouseAge	LotSize	RiverSide	CrimeRate	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	0.389618	0.521869	0.676364	0.113636	0.069170	0.040544	
std	0.204380	0.134627	0.289896	0.233225	0.253994	0.096679	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.267222	0.445392	0.433831	0.000000	0.000000	0.000851	
50%	0.360000	0.507281	0.768280	0.000000	0.000000	0.002812	
75%	0.444444	0.586798	0.938980	0.125000	0.000000	0.041258	
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	

	Industrial	AirQuality	Distance	Highways	Pupils/Teacher	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	0.391378	0.349167	0.242381	0.371713	0.622929	
std	0.251479	0.238431	0.191482	0.378576	0.230313	
min	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.173387	0.131687	0.088259	0.130435	0.510638	
50%	0.338343	0.314815	0.188949	0.173913	0.686170	
75%	0.646628	0.491770	0.369088	1.000000	0.808511	
max	1.000000	1.000000	1.000000	1.000000	1.000000	

	Poverty
count	506.000000
mean	0.301409
std	0.197049
min	0.000000
25%	0.144040
50%	0.265728
75%	0.420116
max	1.000000

6.1 K-means

In Scikit-learn k-means is implemented in the class *KMeans* in module *sklearn.cluster*

```
>>> from sklearn.cluster import KMeans
>>> kmSeg = KMeans(n_clusters=6, n_init=1)
>>> kmSeg.fit(data)
```

The parameter *n_clusters* specifies the number of clusters to create. To perform a k-means analysis in its pure form, the parameter *n_init* has to be set to one. The parameter will be explained below.

To analyze the results, the model object contains the following attributes:

- *cluster_centers_*: Coordinates of cluster centers (array[n_clusters, n_features])
- *labels_*: Labels of each point (array[n_observations])
- *inertia_*: Within Cluster Variance (float)

```
>>> print(kmSeg.cluster_centers_)
[[ 0.44271605  0.53264125  0.35628967 ...,  0.15607581  0.59374432
  0.18642809]
 [ 0.23197133  0.46429145  0.8942477 ...,  1.          0.80851064
  0.48131409]
 [ 0.55383648  0.61536404  0.27058275 ...,  0.10746514  0.41268567
  0.11230684]
 [ 0.42055556  0.54485414  0.79043029 ...,  0.16066576  0.60787899
  0.27316156]
 [ 0.32337778  0.45627132  0.94974253 ...,  0.13826087  0.55382979
  0.41908389]
 [ 0.50679739  0.55898132  0.78424305 ...,  0.37212276  0.53254068
  0.2690235 ]]
```

```
>>> print(kmSeg.labels_)
[5 3 0 ..., 4 1 0]
```

```
>>> labels, counts_labels = np.unique(kmSeg.labels_, return_counts=True)
>>> print("Frequency Clusters:")
>>> from tabulate import tabulate
>>> table = tabulate(np.asarray((labels+1, counts_labels)),
                    tablefmt="grid")

>>> print("\n" + table)
Cluster Frequencys:
+-----+-----+-----+-----+-----+-----+
|  1  |  2  |  3  |  4  |  5  |  6  |
+-----+-----+-----+-----+-----+-----+
| 117 | 124 | 53  | 128 | 50  | 34  |
+-----+-----+-----+-----+-----+-----+

>>> print("Within-Cluster-Variance: ", kmSeg.inertia_)
Within-Cluster-Variance: 99.1849531326
```


One disadvantage of k-means is the random process in the initial phase. Different results of this process can lead to different clustering results. To be able to reproduce the result of your analysis, one can set the *random_state* parameter.

Another way to improve the result of clustering is to use the *n_init* parameter. *n_init* sets the number of time the initialization of the k-means process will be run with different initial seeds. The algorithm then looks for the initial seed that produces the lowest within sum of squares (withinss). That means it tries *n_init* initial random assignments and picks the variant having the lowest initial within sum of squares. Using these initial assignments, the process of k-means is then continued

```
>>> kmSeg = KMeans(n_clusters=6, n_init=10)
>>> kmSeg.fit(data)
>>> labels, counts_labels = np.unique(kmSeg.labels_, return_counts=True)
>>> print("Cluster Frequencies:")
>>> from tabulate import tabulate
>>> table = tabulate(np.asarray((labels+1, counts_labels)),
                                tablefmt="grid")

>>> print("\n" + table)
Cluster Frequencies:
+-----+-----+-----+-----+-----+-----+
|  1  |   2  |   3  |   4  |   5  |   6  |
+-----+-----+-----+-----+-----+-----+
| 53 | 124 | 128 | 34 | 50 | 117 |
+-----+-----+-----+-----+-----+-----+

>>> print("Within-Cluster-Variance: ", kmSeg.inertia_)
Within-Cluster-Variance:  99.1849531326
```

The higher the number of *n_init* the more stable the results are.

The cluster assignments are pulled by using *kmSeg.labels_* and can be appended to the dataframe and stored in a csv file via

```
>>> data_clust = data_ori.copy()    # use copy to avoid creating a pointer
>>> data_clust['Cluster'] = kmSeg.labels_
>>> data_clust.to_csv('Housing2_km.csv')
```

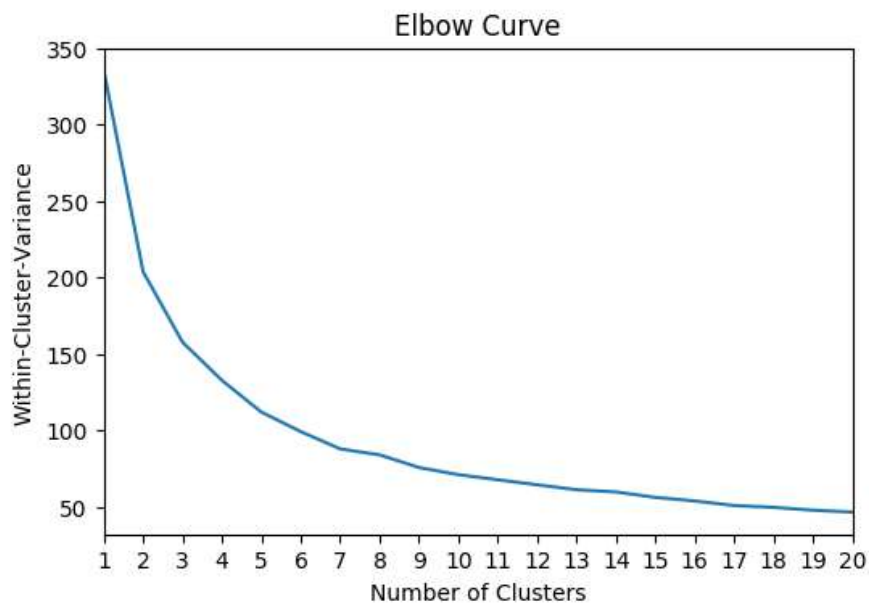
To find the optimal k one can try different values for k and then compare the within sum of squares. Instead of calculating the different k manually, one can use a loop and plot the result

```
>>> wss = []
>>> for k in range(1, 21):
    kmSeg = KMeans(n_clusters=k, n_init=10)
    kmSeg.fit(data)
    wss.append(kmSeg.inertia_)
```

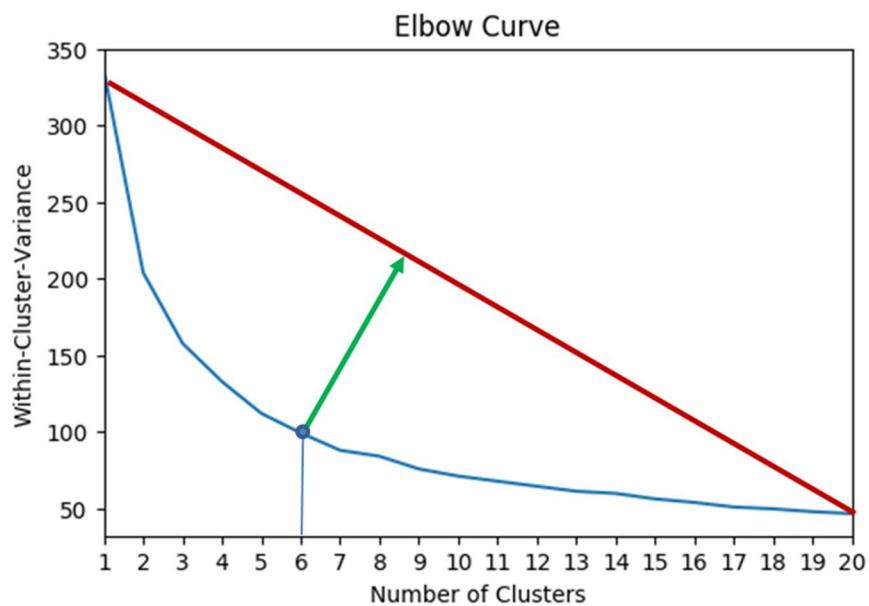
```

>>> import matplotlib.pyplot as plt
>>> plt.plot(range(1, 21), wss)
>>> plt.xlim(1,20)
>>> plt.xticks(range(1, 21))
>>> plt.xlabel('Number of Clusters')
>>> plt.ylabel('Within-Cluster-Variance')
>>> plt.title('Elbow Curve')
>>> plt.show()

```



As the graph shows, it is not always possible to visually identify the optimal cluster number according to the elbow criterion. To support the decision, we can calculate the distances from the points on the curve to a straight line linking the first and the last point on the curve



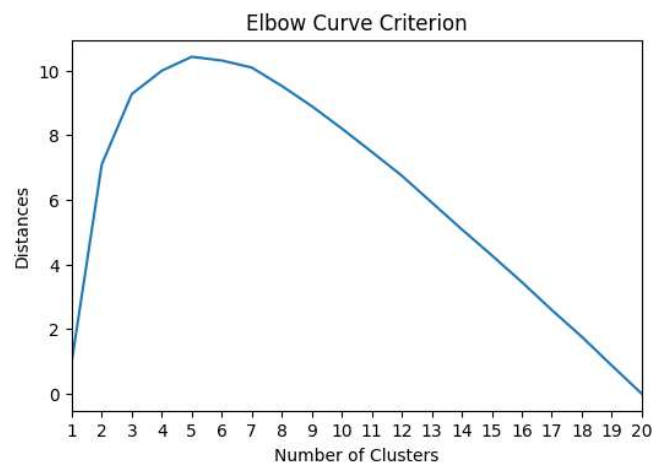
The cluster number with the largest distance is then chosen as the one with the strongest kink. To calculate the line and the distances, we use

```
>>> import math
>>> class Point:
    def __init__(self, initx, inity):
        self.items_list = []
        self.x = initx
        self.y = inity
    def distance_to_line(self, p1, p2):
        x_diff = p2.x - p1.x
        y_diff = p2.y - p1.y
        num = abs(y_diff*self.x-x_diff*self.y + p2.x*p1.y-p2.y*p1.x)
        den = math.sqrt(y_diff**2 + x_diff**2)
        return num / den

>>> distances = []
>>> for k in range(1, 21):
    p1 = Point(initx=1,inity=wss[0])
    p2 = Point(initx=21,inity=wss[19])
    p = Point(initx=k+1,inity=wss[k-1])
    distances.append(p.distance_to_line(p1,p2))
```

Now, we can visualize the distances

```
>>> plt.plot(range(1, 21), distances)
>>> plt.xlim(1,20)
>>> plt.xticks(range(1, 21))
>>> plt.xlabel('Number of Clusters')
>>> plt.ylabel('Distance')
>>> plt.title('Elbow Curve')
>>> plt.show()
```



and choose the optimal number of clusters by identifying the number corresponding to the longest line

```
>>> opt_k = np.argmax(distances) + 1
>>> print('Optimal Cluster Number =', opt_k)
Optimal Cluster Number = 5
```

6.2 Hierarchical Cluster Analysis

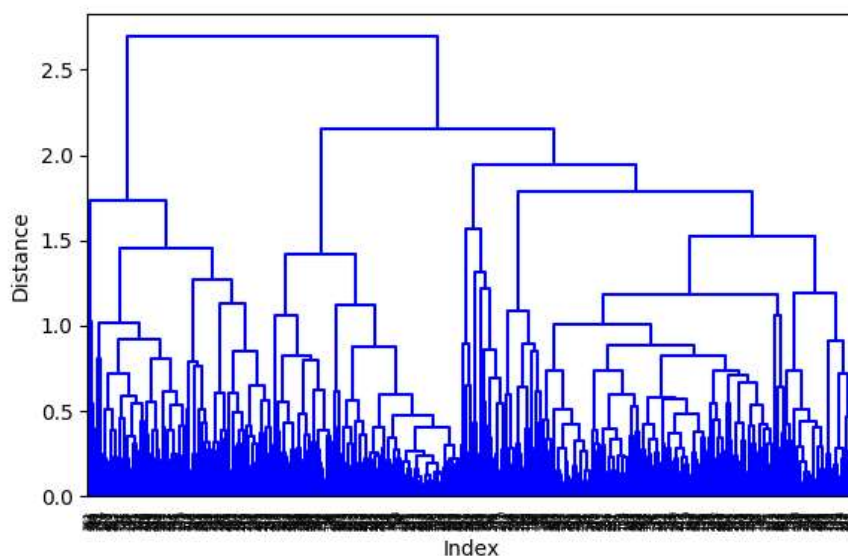
Hierarchical clustering builds a hierarchy from the bottom-up, and doesn't require to specify the number of clusters beforehand. Instead, a dendrogram can be created first which is a tree diagram plotting of distances between each merging of objects. A dendrogram can be created using the *scipy* library

```
>>> from scipy.cluster.hierarchy import dendrogram, linkage
>>> linkage_matrix = linkage(data, method='complete', metric='euclidean')
```

The *linkage* function performs a hierarchical agglomerative clustering and returns the linkage matrix containing the distances between each merging of objects. Available distance metrics are 'euclidean' (default), 'cityblock', 'hamming', and 'cosine'. Linkage methods are 'single', 'complete', 'average', 'weighted', 'centroid', and 'ward'.

The dendrogram can be visualized via

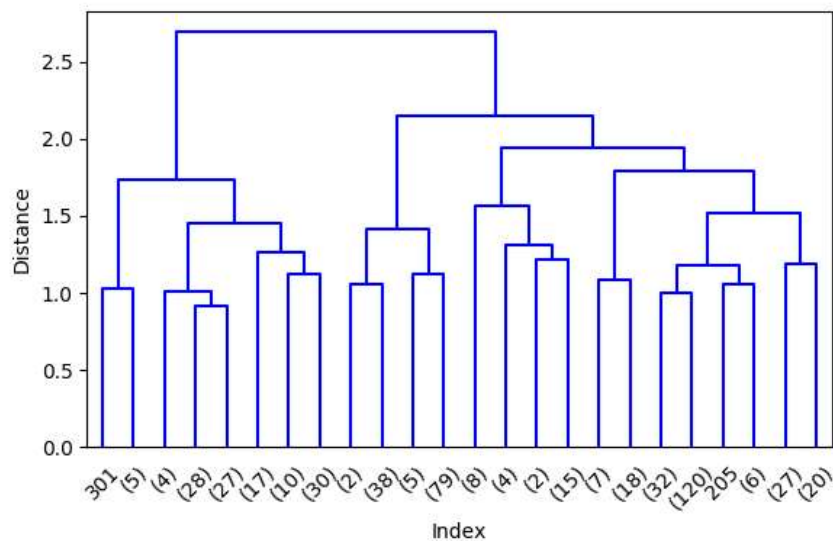
```
>>> figure = plt.figure()
>>> dendrogram(linkage_matrix, color_threshold=0)
>>> plt.xlabel('Index')
>>> plt.ylabel('Distance')
>>> plt.tight_layout()
>>> plt.show()
```



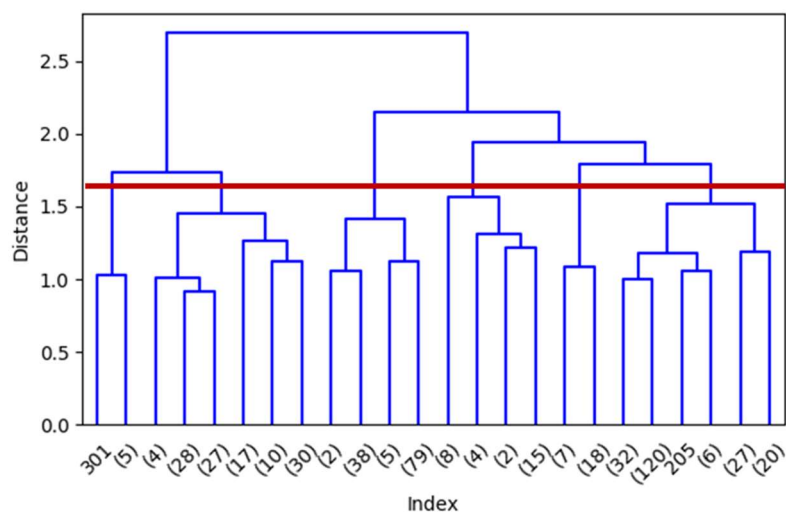
To inspect only the last p merges, we can modify the dendrogram

```
>>> figure = plt.figure()
>>> dendrogram(linkage_matrix, color_threshold=0, truncate_mode='lastp',
                p=24)

>>> plt.xlabel('Index')
>>> plt.ylabel('Distance')
>>> plt.tight_layout()
>>> plt.show()
```



In a dendrogram a horizontal line visualizes the merging of two objects. In a dendrogram the widths of the horizontal lines give an impression about the dissimilarity of the merging object. Thus, a good cluster number might be at a point from where the width of the following horizontal lines is significantly smaller in length. The red line in the graph below shows such a point



Counting the points that cut this line might be a good answer for the number of clusters the data can have. It is the number 6 in this case.

To perform an agglomerative clustering with a predetermined number of clusters, we can use the *AgglomerativeClustering* class from the module *sklearn.cluster*

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> hcSeg = AgglomerativeClustering(n_clusters=6, linkage='complete',
                                     affinity='euclidean')
>>> hcSeg.fit(data)
```

Parameter values are `affinity = {'euclidean', 'l1', 'l2', 'manhattan', 'cosine'}` and `linkage = {'ward', 'complete', 'average'}`.

Finally, we can count the labels and print a frequency table

```
>>> print(hcSeg.labels_)
[0 1 1 ..., 1 2 3]
>>> labels, counts_labels = np.unique(hcSeg.labels_, return_counts=True)
>>> print("Cluster Frequencies:")
>>> from tabulate import tabulate
>>> table = tabulate(np.asarray((labels+1, counts_labels)),
                    tablefmt="grid")

>>> print("\n" + table)
Cluster Frequencies:
+-----+-----+-----+-----+-----+-----+
|  1  |   2  |   3  |   4  |   5  |   6  |
+-----+-----+-----+-----+-----+-----+
| 29  | 206  | 124  | 116  |  25  |   6  |
+-----+-----+-----+-----+-----+-----+
```

6.3 Self-organizing Maps

Self-organizing Maps are not supported by Scikit-learn. There exist different libraries for Python. The most useful library is *Sompy*. Because the original version is not fully compatible with python 3.6 and above, the file “SOMPY-3070e7d747a3e4d7b4135bbffd1311dc0411b1ae.zip” must be downloaded from “<https://github.com/altermarkive/SOMPY/tree/3070e7d747a3e4d7b4135bbffd1311dc0411b1ae>”. Unzip the file and run

```
python setup.py install
```

from the python console and the unzipped directory.

After installation, the library can be used. Sompy does not work with pandas dataframes. Thus, a dataframe must be converted into a numpy array first. Because Sompy has a built-in normalization, we use the original dataframe

```
>>> data_np = np.array(data_ori)
```

Next the SOM can be initialized and trained

```
>>> import sompy as som
>>> mapsize = [20,20]
>>> sommodel = som.SOMFactory.build(data_np, mapsize,
                                     initialization='pca',
                                     neighborhood='gaussian',
                                     component_names=list(data))

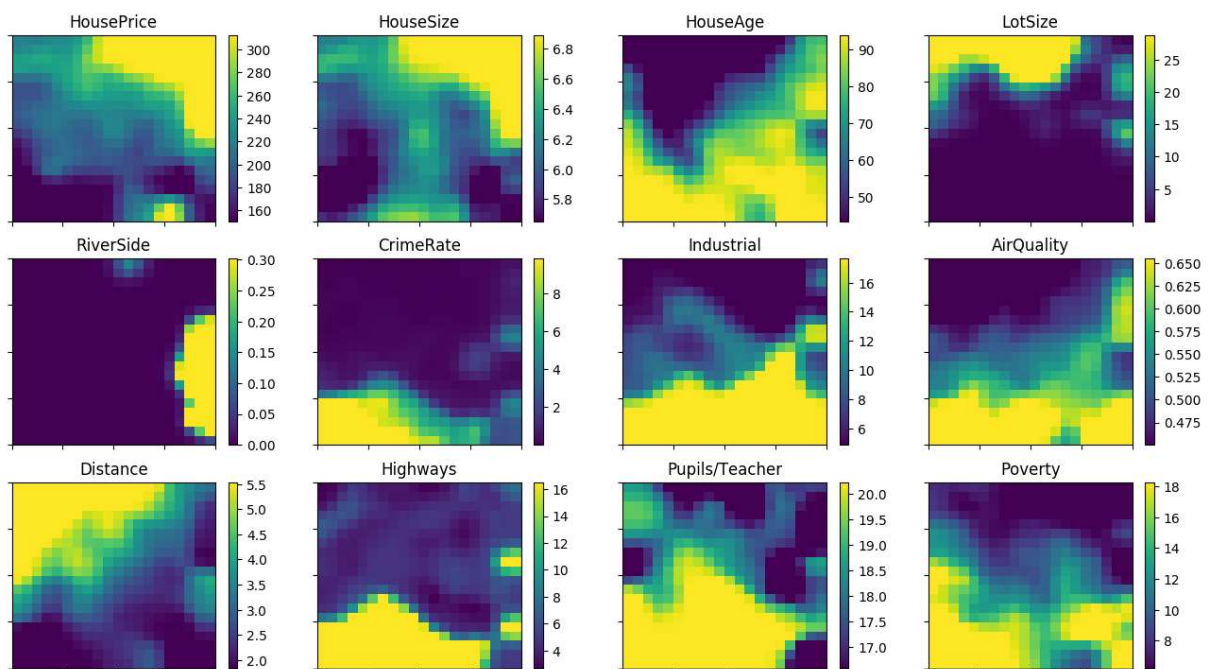
>>> sommodel.train(verbose=None)
```

The parameters are:

- **data:** data to be clustered, represented as an Numpy array of n rows as inputs and m cols as features.
- **neighborhood:** neighborhood object calculator. Options are: {gaussian, bubble}
- **mapsize:** tuple/list defining the dimensions of the som. If a single number is provided it is considered as the number of nodes.
- **initialization:** method to be used for initialization of the SOM. Options are: {pca, random}
- **component_names:** names of the features

To visualize the results, Sompy provides different types of plots. The first visualization to make note of are the heatmaps of the features

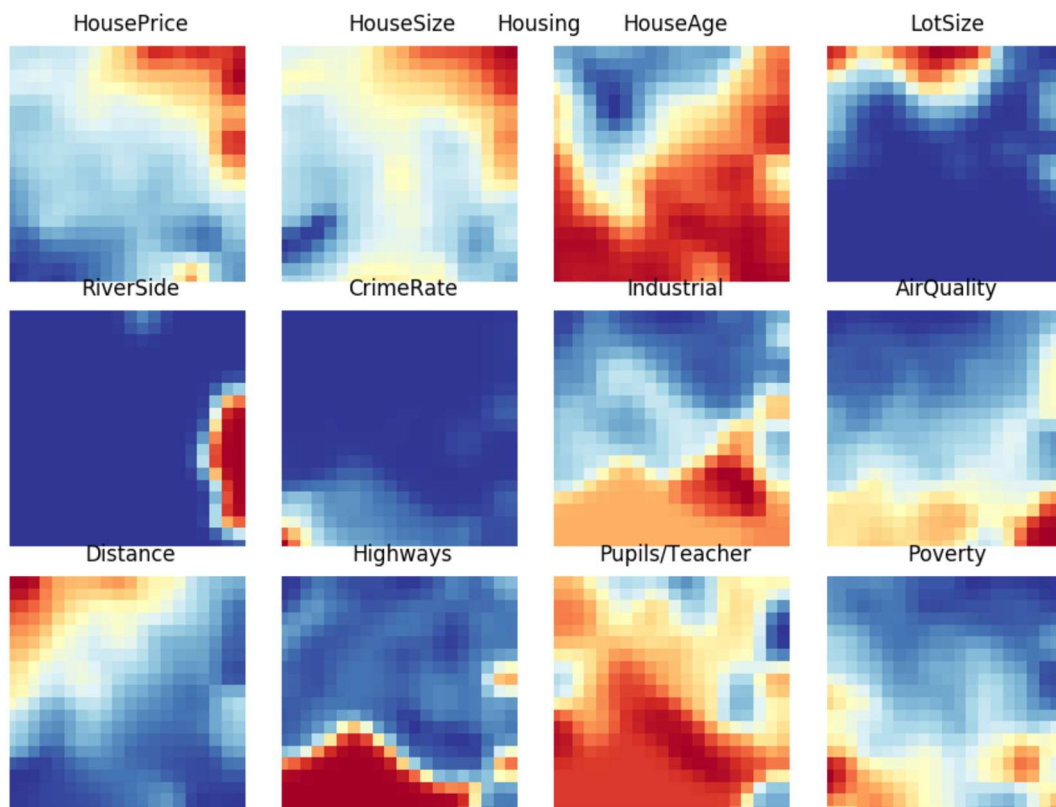
```
>>> from sompy.visualization.mapview import View2D
>>> view2D = View2D(10,10,"Housing",text_size=10)
>>> view2D.show(sommodel, col_sz=4, which_dim="all", desnormalize=True)
```



Each heatmap represents the intensity of a single feature or data column as learned by the SOM grid. The heatmap represents the values of a feature over the map. It is discretized and each 'block' in each subfigure is a unique neuron, the neurons have the same position across all subfigures. In general, similar heatmaps of different features represent correlation of features, dissimilarity represents negligible correlation and inversion represents anti-correlation.

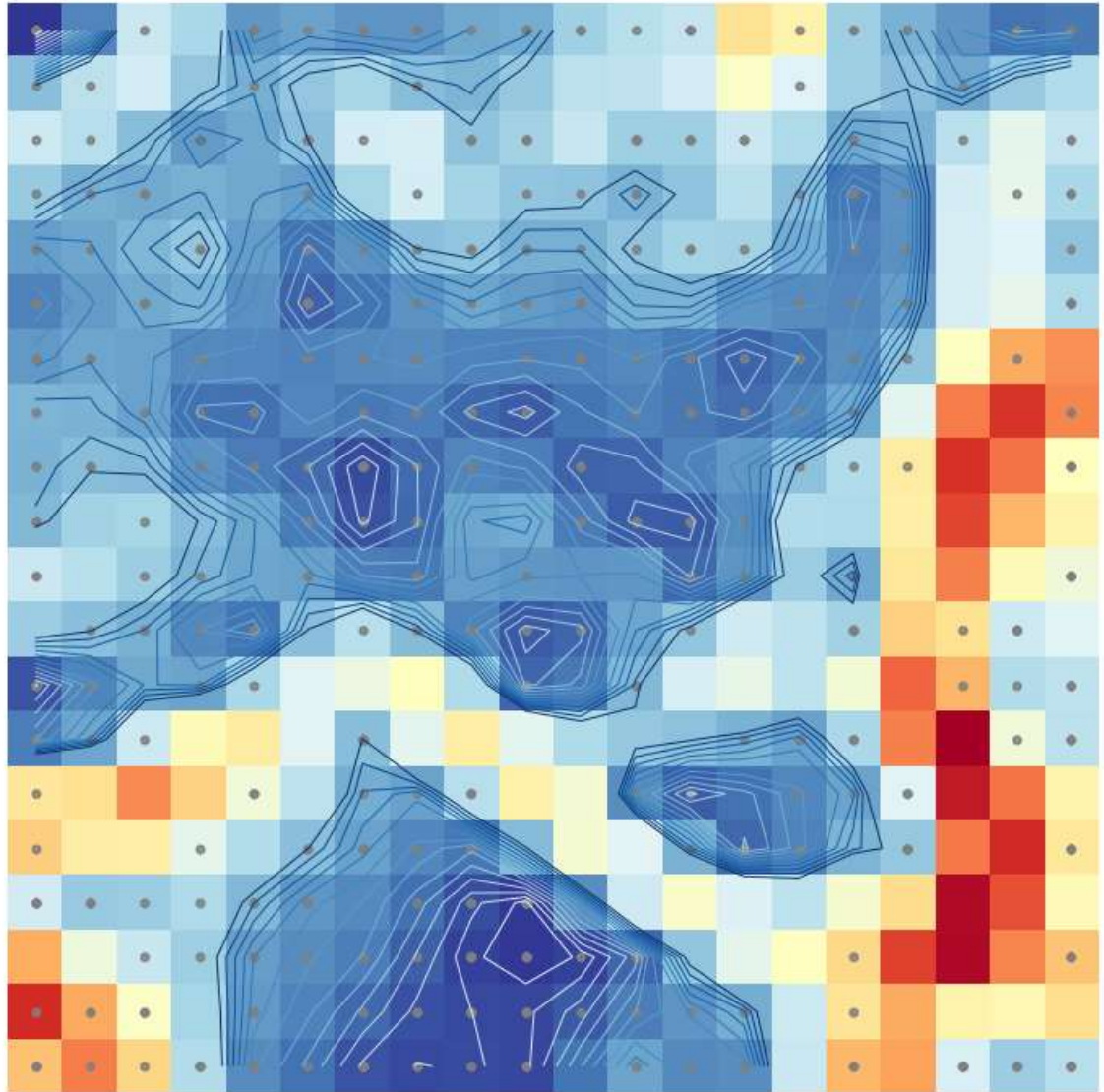
To create the heatmaps without the scales for the colors, the *View2DPacked* function can be used

```
>>> from sompy.visualization.mapview import View2DPacked
>>> view2D_pk = View2DPacked(10,10,"Housing",text_size=10)
>>> view2D_pk.show(sommodel, col_sz=4, which_dim="all")
```



The next visualization we can do is called the U-matrix. The U-matrix is also a heatmap but can be interpreted like a topographic map (a map that shows the elevation contours across a region) across our SOM grid. The 'hills' in the U-matrix represent large distances between neighbors and vice versa for valleys

```
>>> from sompy.visualization.umatrix import UMatrixView
>>> umat = UMatrixView(width=10,height=10,title='U-matrix')
>>> umat.show(sommodel)
```

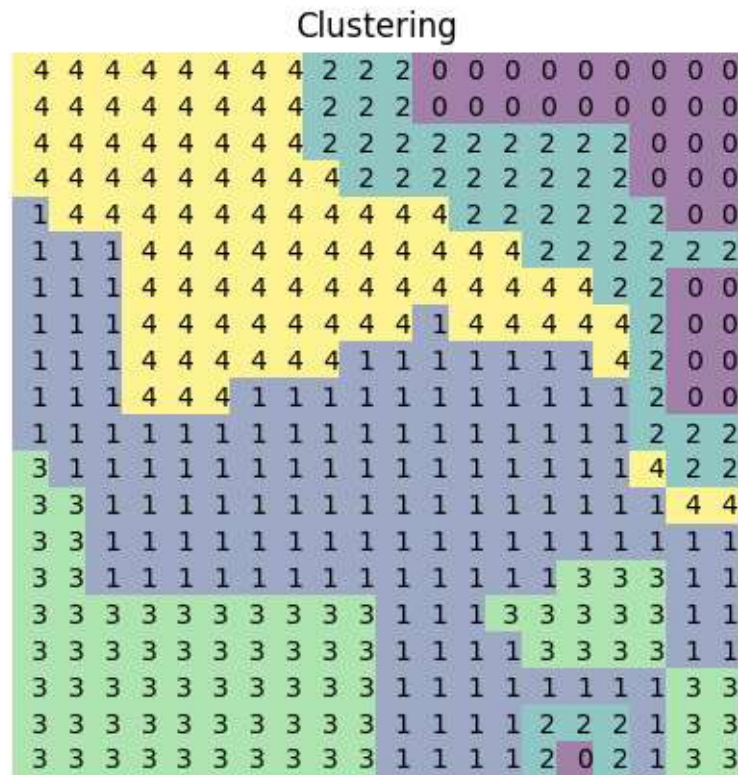
The U-matrix gives us a sense of the landscape of the SOM grid, and what the neighborhoods of the neurons are. It helps us visually see what potential clustering there is in the SOM grid itself.

Now, a clustering can be performed. Subject of the clustering are the nodes (neurons) of the map. Sompy has a built-in function *cluster* which uses the k-means function of Scikit-learn. The only parameter is the number of clusters and the function returns the cluster labels: The number of clusters can be estimated from the U-matrix. Here, we decide to use create 5 clusters

```
>>> labels = sommodel.cluster(5)
>>> print(labels)
[4 4 4 ..., 1 3 3]
>>> print(labels.shape)
(400,)
```

To visualize the assignment of the clusters to the nodes on the map, we can create a so-called hitmap

```
>>> from sompy.visualization.hitmap import HitMapView
>>> hits = HitMapView(20,20,"Clustering",text_size=10)
>>> a=hits.show(sommodel)
```



Next, we can map the real data objects to their best matching nodes (bmu)

```
>>> bmus = sommodel.project_data(data_ori)
>>> print(bmus)
[259 192 120 ..., 313 320 107]
>>> print(bmus.shape)
(506,)
```

and use their cluster assignments to assign cluster numbers to the data objects. Because Sompy counts the cluster numbers starting with 0, we add a one to be conform with the counting scheme from the former chapters

```
>>> clusts = labels[bmus]+1
>>> print(clusts)
[4 1 1 ..., 3 3 4]
>>> print(clusts.shape)
(506,)
```

We can count the labels and print a frequency table

```
>>> clabels, counts_labels = np.unique(clusts, return_counts=True)
>>> print("Cluster Frequencies:")
>>> from tabulate import tabulate
>>> table = tabulate(np.asarray((clabels, counts_labels)),
                                tablefmt="grid")

>>> print("\n" + table)
Cluster Frequencies:
+-----+-----+-----+-----+-----+
|  1  |  2  |  3  |  4  |  5  |
+-----+-----+-----+-----+-----+
| 159 |  63 | 129 | 120 |  35 |
+-----+-----+-----+-----+-----+
```

Finally, the assignments can be added to the original (not normalized) data and stored in a csv file

```
>>> data_clust = pd.DataFrame(data_ori)
>>> data_clust['Cluster'] = pd.DataFrame(clusts)
>>> data_clust.to_csv('clusterdata.csv')
```

6.4 t-SNE

The t-SNE algorithm is implemented in class *TSNE* in *sklearn.manifold*

```
>>> from sklearn.manifold import TSNE
>>> tsne_model = TSNE(n_components=2, perplexity=20, random_state=0)
>>> tsne_results = tsne_model.fit_transform(data_ori)
```

TSNE has the following parameters:

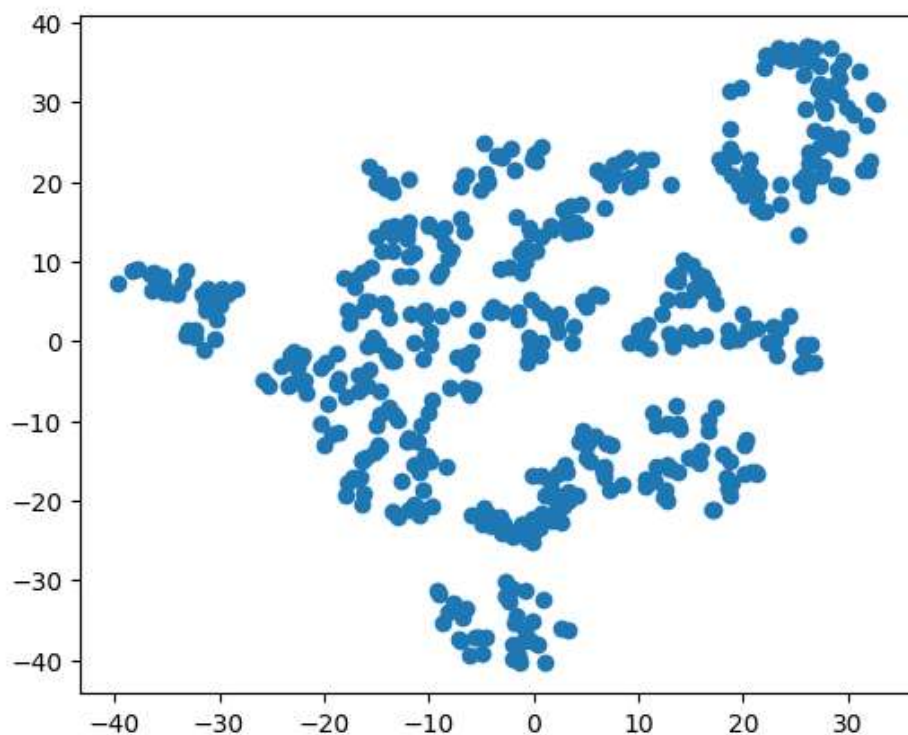
- **data:** input data, represented as dataframe array of n rows, as inputs and m cols as input features.
- **n_components:** dimensions of the output space (default=2).
- **perplexity:** This value effectively controls how many nearest neighbors are taken into account when constructing the embedding in the low-dimensional space (default=30). This is a crucial parameter (see <http://distill.pub/2016/misread-tsne>).
- **learning_rate:** The learning rate for t-SNE is usually in the range [10.0, 1000.0] (default: 200.0). If the learning rate is too high, the data may look like a ‘ball’ with any point approximately equidistant from its nearest neighbours. If the learning rate is too low, most points may look compressed in a dense cloud with few outliers.
- **n_iter:** Number of iterations (default: 1000)

The result of the `fit_transform` function are the coordinates of the objects in the n-dimensional map

```
>>> print(tsne_results)
[[ 3.82877496  1.83875984]
 [-11.4038996 -0.2693186 ]
 [ 13.28204502 -0.20257933]
 ...,
 [-11.02187217 -20.94664243]
 [ 26.02680503  18.33161713]
 [ 11.78928785 -10.57281252]]
```

They can be used to plot the map

```
>>> plt.figure(figsize=(6, 5))
>>> plt.scatter(tsne_results[:, 0], tsne_results[:, 1])
>>> plt.show()
```



The coordinates can now be the base for applying the clustering algorithm. Here, we apply K-means. At first, we have to normalize the coordinates. Next, we try different cluster numbers and use the elbow criterion to select the best k

```
>>> tsne_data = (tsne_results-tsne_results.min())/(tsne_results.max()
                                                    -tsne_results.min())

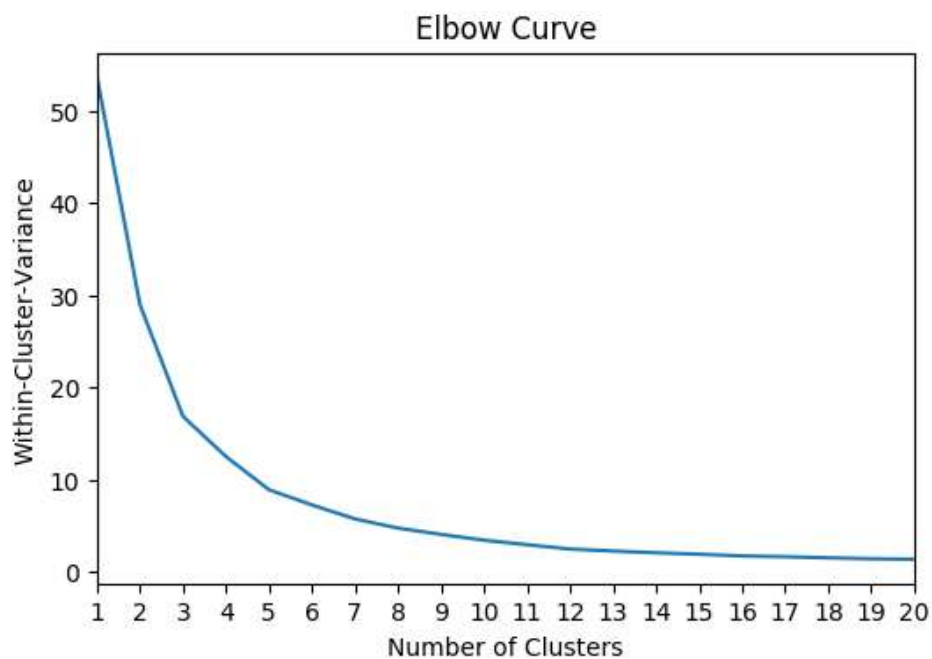
>>> from sklearn.cluster import KMeans

>>> wss = []
```

```

>>> for k in range(1, 21):
    tsneSeg = KMeans(n_clusters=k, n_init=10)
    tsneSeg.fit(tsne_data)
    wss.append(tsneSeg.inertia_)
>>> plt.plot(range(1, 21), wss)
>>> plt.xlim(1,20)
>>> plt.xticks(range(1, 21))
>>> plt.xlabel('Number of Clusters')
>>> plt.ylabel('Within-Cluster-Variance')
>>> plt.title('Elbow Curve')
>>> plt.show()

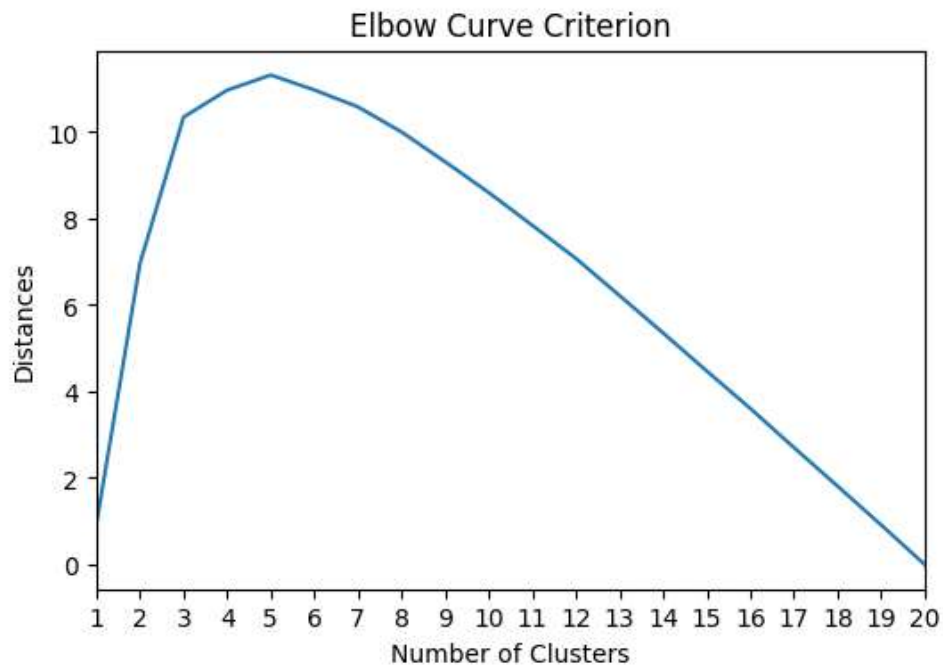
```



```

>>> distances = []
>>> for k in range(1, 21):
    p1 = Point(initx=1, inity=wss[0])
    p2 = Point(initx=21, inity=wss[19])
    p = Point(initx=k+1, inity=wss[k-1])
    distances.append(p.distance_to_line(p1,p2))
>>> plt.plot(range(1, 21), distances)
>>> plt.xlim(1,20)
>>> plt.xticks(range(1, 21))
>>> plt.xlabel('Number of Clusters')
>>> plt.ylabel('Distances')
>>> plt.title('Elbow Curve Criterion')
>>> plt.show()

```



```
>>> opt_k = np.argmax(distances) + 1
>>> print('Optimal Cluster Number =', opt_k)
Optimal Cluster Number = 5
```

We use the best k to perform the cluster analysis on the coordinates

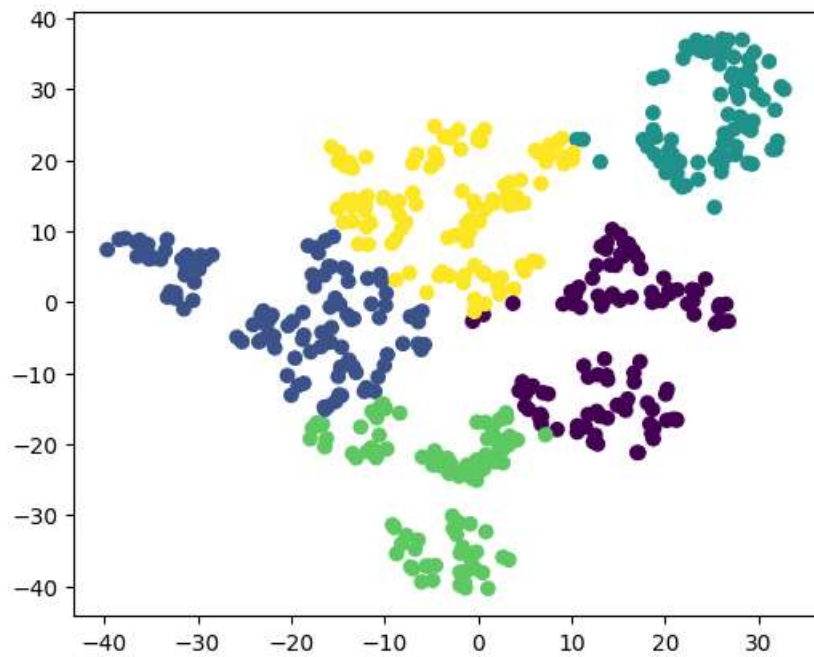
```
>>> tsneSeg = KMeans(n_clusters=opt_k, n_init=10)
>>> tsneSeg.fit(tsne_data)
>>> labels, counts_labels = np.unique(tsneSeg.labels_, return_counts=True)
>>> print("Cluster Frequencys:")
>>> from tabulate import tabulate
>>> table = tabulate(np.asarray((labels+1, counts_labels)),
                    tablefmt="grid")

>>> print("\n" + table)
Cluster Frequencys:
```

```
+-----+-----+-----+-----+
|  1  |  2  |  3  |  4  |  5  |
+-----+-----+-----+-----+
| 107 | 113 | 82  | 95  | 109 |
+-----+-----+-----+-----+
```

and color the points in the map according to the cluster memberships

```
>>> plt.figure(figsize=(6, 5))
>>> plt.scatter(tsne_results[:, 0], tsne_results[:, 1],
               c=tsneSeg.labels_)
>>> plt.show()
```



To create a 3-dimensional map, the parameter *n_components* must be set to 3. The remaining procedure is the same as above with the difference that the result matrix has now 3 axis

```
>>> tsne3dmodel = TSNE(n_components=3, perplexity=20, random_state=0)
>>> tsne3d_results = tsne3dmodel.fit_transform(data_ori)
>>> print(tsne3d_results)
[[ 4.75832576 13.64922197  0.34242926]
 [ -4.14555928  0.5245007  4.55680377]
 [ 15.66017675 -7.19635628 -2.22184973]
 ...,
 [-10.03253514 -17.28221186 -2.18456374]
 [ -2.83826115 17.68304295 -4.22097819]
 [ 9.94497659  2.78642731 -19.3615228  ]]
```

We cluster the results using k=5

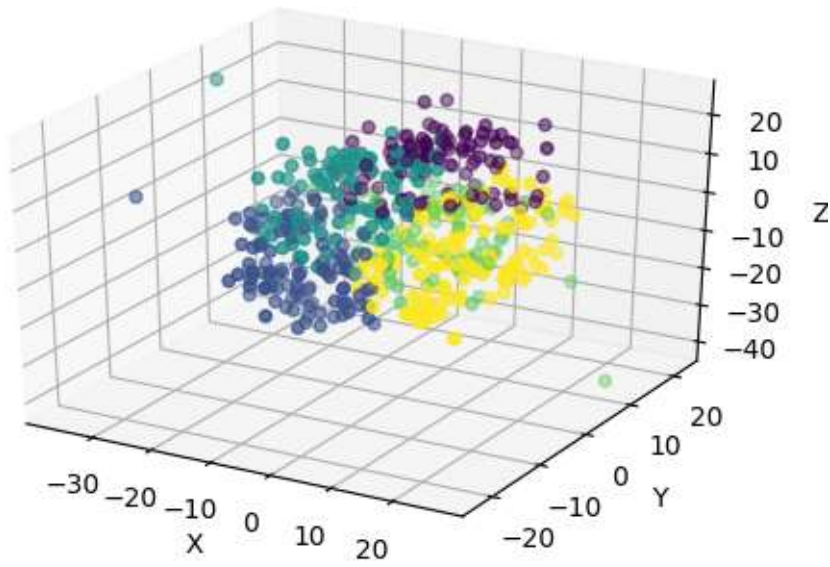
```
>>> tsne_data = (tsne3d_results-tsne3d_results.min())
                /(tsne3d_results.max()-tsne3d_results.min())
>>> tsneSeg = KMeans(n_clusters=5, n_init=10)
>>> tsneSeg.fit(tsne_data)
```

and plot the results as 3d-scatter plot

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> x = tsne3d_results[:, 0]
>>> y = tsne3d_results[:, 1]
```



```
>>> z = tsne3d_results[:, 2]
>>> ax.scatter(x, y, z, c=tsneSeg.labels_)
>>> ax.set_xlabel('X')
>>> ax.set_ylabel('Y')
>>> ax.set_zlabel('Z')
>>> plt.show()
```



7 Association Analysis¹

In this chapter, we use a case of from market basket analysis. The data for this article comes from the UCI Machine Learning Repository and represents transactional data from a UK retailer from 2010-2011. Each line represents one item purchased in different transactions which are indicated by the InvoiceNo. The file "OnlineRetail.xlsx" looks in the following way

	A	B	C	D	E	F	G	H
1	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
2	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	01.12.2010 08:26	2,55	17850	United Kingdom
3	536365	71053	WHITE METAL LANTERN	6	01.12.2010 08:26	3,39	17850	United Kingdom
4	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	01.12.2010 08:26	2,75	17850	United Kingdom
5	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	01.12.2010 08:26	3,39	17850	United Kingdom
6	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	01.12.2010 08:26	3,39	17850	United Kingdom
7	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	01.12.2010 08:26	7,65	17850	United Kingdom
8	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	01.12.2010 08:26	4,25	17850	United Kingdom
9	536366	22633	HAND WARMER UNION JACK	6	01.12.2010 08:28	1,85	17850	United Kingdom
10	536366	22632	HAND WARMER RED POLKA DOT	6	01.12.2010 08:28	1,85	17850	United Kingdom
11	536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	01.12.2010 08:34	1,69	13047	United Kingdom
12	536367	22745	POPPY'S PLAYHOUSE BEDROOM	6	01.12.2010 08:34	2,1	13047	United Kingdom

To perform the association analysis, we use the MLxtend library. First, we have to read the data from the Excel file and store it in a Pandas dataframe

```
>>> import os
>>> import pandas as pd
>>> os.chdir("D:/Dropbox/Lehre/Digital Analytics/Excercises/Association")
>>> df = pd.read_excel('OnlineRetail.xlsx')
>>> df.head()
```

	InvoiceNo	StockCode	Description	Quantity	\
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	
1	536365	71053	WHITE METAL LANTERN	6	
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	

	InvoiceDate	UnitPrice	CustomerID	Country
0	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
1	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
2	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
3	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
4	2010-12-01 08:26:00	3.39	17850.0	United Kingdom

There is a little cleanup, we need to do. First, some of the descriptions have spaces that need to be removed. We'll also drop the rows that don't have invoice numbers and remove the credit transactions (those with invoice numbers containing C)

```
>>> df['Description'] = df['Description'].str.strip()
>>> df.dropna(axis=0, subset=['InvoiceNo'], inplace=True)
>>> df['InvoiceNo'] = df['InvoiceNo'].astype('str')
>>> df = df[~df['InvoiceNo'].str.contains('C')]
```

¹ based on Introduction to Market Basket Analysis in Python by Chris Moffitt, URL: <http://pbpython.com/market-basket-analysis.html>

After the cleanup, we need to consolidate the items into one transaction per row with each product as column one hot encoded. For the apriori function in mlxtend the required data structure is a pandas dataframe having the encoded format. The allowed values for the columns are either 0/1 or True/False. For example

	Apple	Bananas	Beer	Chicken	Milk	Rice	Oil
0	1	0	0	1	1	0	1
1	1	0	0	1	0	0	1
2	1	0	0	1	0	0	0
3	1	1	1	0	0	0	0
4	0	0	0	1	1	1	1
5	0	0	0	1	0	1	1
6	0	0	0	1	0	1	0
7	1	1	1	0	0	0	0

To reduce the calculation time needed, we are now only extracting the sales for France

```
>>> basket = (df[df['Country'] == "France"]
               .groupby(['InvoiceNo', 'Description'])['Quantity']
               .sum().unstack().reset_index().fillna(0)
               .set_index('InvoiceNo'))
```

There are a lot of zeros and some ones in the data but all in floating point format. Additionally, we also need to make sure any positive values are converted to a 1 and anything less the 0 is set to 0. This step will complete the one hot encoding of the data and remove the postage column (since that charge is not one we wish to explore)

```
>>> def encode_units(x):
    if x <= 0:
        return 0
    if x >= 1:
        return 1

>>> basket_sets = basket.applymap(encode_units)
>>> basket_sets.drop('POSTAGE', inplace=True, axis=1)
```

Now that the data is structured properly, we can generate frequent item sets that have a support of at least 7% for examples

```
>>> from mlxtend.frequent_patterns import apriori
>>> frequent_itemsets = apriori(basket_sets, min_support=0.07,
                                use_colnames=True)
```

The final step is to generate the rules with their corresponding support, confidence and lift

```
>>> from mlxtend.frequent_patterns import association_rules
```

```
>>> rules = association_rules(frequent_itemsets, metric="lift",
                              min_threshold=1)

>>> rules.head()
```

	antecedents	consequents
0	(PLASTERS IN TIN SPACEBOY)	(PLASTERS IN TIN CIRCUS PARADE)
1	(PLASTERS IN TIN CIRCUS PARADE)	(PLASTERS IN TIN SPACEBOY)
2	(SET/6 RED SPOTTY PAPER CUPS)	(SET/20 RED RETROSPOT PAPER NAPKINS)
3	(SET/20 RED RETROSPOT PAPER NAPKINS)	(SET/6 RED SPOTTY PAPER CUPS)
4	(PLASTERS IN TIN SPACEBOY)	(PLASTERS IN TIN WOODLAND ANIMALS)

	antecedent support	consequent support	support	confidence	lift
0	0.137755	0.168367	0.089286	0.648148	3.849607
1	0.168367	0.137755	0.089286	0.530303	3.849607
2	0.137755	0.132653	0.102041	0.740741	5.584046
3	0.132653	0.137755	0.102041	0.769231	5.584046
4	0.137755	0.170918	0.104592	0.759259	4.442233

	leverage	conviction
0	0.066092	2.363588
1	0.066092	1.835747
2	0.083767	3.345481
3	0.083767	3.736395
4	0.081047	3.443878

Besides support, confidence and lift two additional measures are calculated here:

- Leverage measures the difference between the frequency of co-occurrence of antecedent and consequent as the independent frequencies of each of antecedent and consequent. In other words, leverage measures the proportion of additional cases covered by both antecedent and consequent above those expected if antecedent and consequent were independent of each other.

Definition: $\text{leverage}(A \rightarrow B) = \text{support}(A \rightarrow B) - \text{support}(A) * \text{support}(B)$

- Conviction is a measure of the implication and has value 1 if items are unrelated. It can be interpreted as the ratio of the expected frequency that X occurs without Y.

Definition: $\text{conviction} = [1 - \text{support}(B)] / [1 - \text{confidence}(A \rightarrow B)]$

After building the frequent items using *apriori* and building the rules with *association_rules*, we can now filter the dataframe *rules* using standard Pandas code.

In this case, look for a large lift (6) and high confidence (.8)

```
>>> print(rules[ (rules['lift'] >= 6) & (rules['confidence'] >= 0.8) ])
```

	antecedants
6	(SET/6 RED SPOTTY PAPER CUPS, SET/6 RED SPOTTY...
7	(SET/6 RED SPOTTY PAPER CUPS, SET/20 RED RETRO...
8	(SET/6 RED SPOTTY PAPER PLATES, SET/20 RED RET...
14	(ALARM CLOCK BAKELIKE RED)
15	(ALARM CLOCK BAKELIKE GREEN)
18	(SET/6 RED SPOTTY PAPER CUPS)
19	(SET/6 RED SPOTTY PAPER PLATES)
20	(SET/6 RED SPOTTY PAPER PLATES)

	consequents	antecedent support \
6	(SET/20 RED RETROSPOT PAPER NAPKINS)	0.122449
7	(SET/6 RED SPOTTY PAPER PLATES)	0.102041
8	(SET/6 RED SPOTTY PAPER CUPS)	0.102041
14	(ALARM CLOCK BAKELIKE GREEN)	0.094388
15	(ALARM CLOCK BAKELIKE RED)	0.096939
18	(SET/6 RED SPOTTY PAPER PLATES)	0.137755
19	(SET/6 RED SPOTTY PAPER CUPS)	0.127551
20	(SET/20 RED RETROSPOT PAPER NAPKINS)	0.127551

	consequent support	support	confidence	lift	leverage	conviction
6	0.132653	0.099490	0.812500	6.125000	0.083247	4.625850
7	0.127551	0.099490	0.975000	7.644000	0.086474	34.897959
8	0.137755	0.099490	0.975000	7.077778	0.085433	34.489796
14	0.096939	0.079082	0.837838	8.642959	0.069932	5.568878
15	0.094388	0.079082	0.815789	8.642959	0.069932	4.916181
18	0.127551	0.122449	0.888889	6.968889	0.104878	7.852041
19	0.137755	0.122449	0.960000	6.968889	0.104878	21.556122
20	0.132653	0.102041	0.800000	6.030769	0.085121	4.336735

In looking at the rules, it seems that the green and red alarm clocks are purchased together and the red paper cups, napkins and plates are purchased together in a manner that is higher than the overall probability would suggest.

At this point, you may want to look at how much opportunity there is to use the popularity of one product to drive sales of another. For instance, we can see that we sell 340 Green Alarm clocks but only 316 Red Alarm Clocks so maybe we can drive more Red Alarm Clock sales through recommendations?

```
>>> print(basket['ALARM CLOCK BAKELIKE GREEN'].sum())
340.0
>>> print(basket['ALARM CLOCK BAKELIKE RED'].sum())
316.0
```

What is also interesting is to see how the combinations vary by country of purchase. Let's check out what some popular combinations might be in Germany

```
>>> basket2 = (df[df['Country'] == "Germany"]
               .groupby(['InvoiceNo', 'Description'])['Quantity']
               .sum().unstack().reset_index().fillna(0)
               .set_index('InvoiceNo'))
>>> basket_sets2 = basket2.applymap(encode_units)
>>> basket_sets2.drop('POSTAGE', inplace=True, axis=1)
>>> frequent_itemsets2 = apriori(basket_sets2, min_support=0.05,
                                use_colnames=True)
>>> rules2 = association_rules(frequent_itemsets2, metric="lift",
                              min_threshold=1)
>>> print(rules2[ (rules2['lift'] >= 4) & (rules2['confidence'] >= 0.5)])
```

	antecedants	consequents \
0	(RED RETROSPOT CHARLOTTE BAG)	(WOODLAND CHARLOTTE BAG)
10	(PLASTERS IN TIN SPACEBOY)	(PLASTERS IN TIN WOODLAND ANIMALS)
12	(PLASTERS IN TIN CIRCUS PARADE)	(PLASTERS IN TIN WOODLAND ANIMALS)

	antecedent support	consequent support	support	confidence	lift	\
0	0.070022	0.126915	0.059081	0.843750	6.648168	
10	0.107221	0.137856	0.061269	0.571429	4.145125	
12	0.115974	0.137856	0.067834	0.584906	4.242887	
	leverage	conviction				
0	0.050194	5.587746				
10	0.046488	2.011670				
12	0.051846	2.076984				