Machine_Learning_Prototyp

November 14, 2018

Prototypisierung des theoretischen Konzeptes

Das Vorgehen der Prototypisierung richtet sich an die Phasen des CRISP-DM Prozessmodells. Die beiden Phasen Business Understanding und Deployment werden dabei nicht für die praktische Umsetzung berücksichtigt, da diese Phasen Aufgaben der betrieblichen Praxis enthalten. Somit sind sie für die Demonstration der Funktionen nicht relevant. Die aufgestellten Use Cases sind den Phasen Data Understanding, Data Preperation, Modeling und Evaluation zugeordnet.

Data Understanding

- **UC02 - Datensatz laden**
- **UC03 - Datensatz beschreiben**
- **UC04 - Daten visualisieren**

In [1]:
```
# for dealing with large datasets
import pandas as pd
# for processing multidimensional arrays
import numpy as np
# for importing functions from another notebook
import import_ipynb
# notebook with the logic for data generation
import Data_Generator

# show all visual outputs directly in the Jupyter notebook
%matplotlib inline
```

```
importing Jupyter notebook from Data_Generator.ipynb
```

In [2]:
```
from sklearn.utils import shuffle

# shuffle generated data and load it into a data frame
df = shuffle(Data_Generator.generate_data_set(5000))
# show first five entries of the shuffled data frame
df.head()
```

Out[2]:

	Duration	Region	Km	Stops	Weather	Extreme	Traffic
4074	2.75581	1	2.68609	11		none	26.6477

```
758   4.75395      3  14.3729    48            none  88.5243
1417  4.08024      4  17.9515    98            none  18.2974
4537  4.49769      3  14.6012    69            none  89.1568
3248  7.09394      5  17.5022   138            snow  96.1133
```

Um ein besseres Verständnis über die Daten zu erhalten, erfolgt eine Beschreibung des Datensatzes. Für jede Spalte mit numerischen Werten werden folgende Eigenschaften berechnet und angezeigt:

- Anzahl an Einträgen
- Mittelwert
- Standardabweichung
- Kleinster Wert
- 25%-Perzentil
- Median
- 75%-Perzentil
- Größter Wert

```
In [3]: # convert numeric columns for description
        df['Duration'] = pd.to_numeric(df['Duration'])
        df['Region'] = pd.to_numeric(df['Region'])
        df['Km'] = pd.to_numeric(df['Km'])
        df['Stops'] = pd.to_numeric(df['Stops'])
        df['Traffic'] = pd.to_numeric(df['Traffic'])

        # describe for each numerical column:
        # number of entries,
        # number of unique values,
        # standard deviation,
        # min, max value,
        # lower percentile, upper percentile,
        # median
        df.describe()
```

```
Out[3]:           Duration       Region           Km        Stops      Traffic
        count  5000.000000  5000.000000  5000.000000  5000.000000  5000.000000
        mean      4.765207     3.013200    10.907766    62.333000    50.411542
        std       1.659467     1.419235     5.736581    41.431904    28.609584
        min       0.793058     1.000000     0.011009     1.000000     0.021689
        25%       3.475378     2.000000     5.764785    20.000000    25.877859
        50%       4.772899     3.000000    11.160588    60.000000    50.691117
        75%       6.031007     4.000000    15.763333   101.000000    75.161896
        max       9.067503     5.000000    19.993932   140.000000    99.997895
```

Für Spalten mit nicht numerischen Werten wird folgendes berechnet und angezeigt:

- Anzahl an Einträgen
- Anzahl an einzigartigen Werten
- meist vorkommender Wert

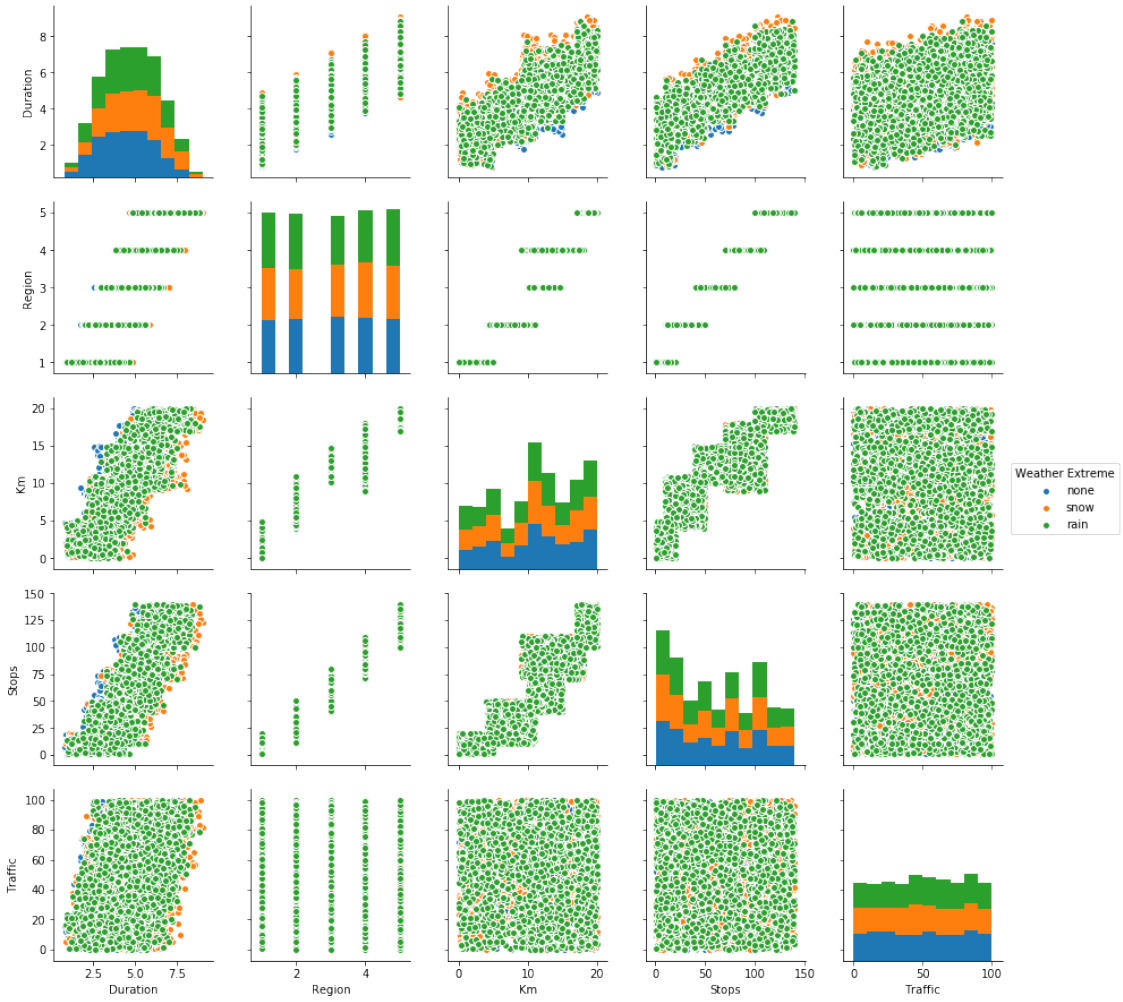
- Häufigkeit des am meisten vorkommenden Wertes

```
In [4]: # describe for non numerical column:
        # number of entries,
        # number of unique values,
        # most common value,
        # most common value frequency
        df.describe(include=[np.object])
```

```
Out[4]:        Weather Extreme
        count             5000
        unique               3
        top               none
        freq              1710
```

Um Einblick in die Verteilung und die Beziehung zwischen den Daten zu erlangen, bietet sich die Visualisierung des Datenbestandes an. Hierfür werden die Beziehungen der einzelnen Merkmalspaare in einer scatter plot matrix dargestellt.

```
In [5]: import seaborn as sns

        # visualize a scatter plot matrix
        # different colors stands for weather values
        splom = sns.pairplot(df, hue = "Weather Extreme")
```



Data Preperation

- **UC05 - Daten transformieren**

Für nicht numerische Werte muss eine Einzelattributstransformation durchgeführt werden, damit die Lernalgorithmen diese in die Berechnung einbeziehen können. Im Fall der generierten Daten muss dies nur für das Wetter durchgeführt werden.

```
In [6]: # transformate non numerical attributes for the learing algorithms
        df = pd.get_dummies(df, columns = ["Weather Extreme"])
        # show first five entries of the preperated data frame
        df.head()
```

```
Out[6]:       Duration  Region         Km  Stops    Traffic  Weather Extreme_none  \
        4074  2.755809       1   2.686089     11  26.647709                     1
        758   4.753949       3  14.372939     48  88.524295                     1
        1417  4.080244       4  17.951528     98  18.297396                     1
```

```
4537  4.497689       3  14.601206     69  89.156768                    1
3248  7.093945       5  17.502232    138  96.113309                    0

      Weather Extreme_rain  Weather Extreme_snow
4074                     0                     0
758                      0                     0
1417                     0                     0
4537                     0                     0
3248                     0                     1
```

Modeling

- **UC06 - Testdesign erstellen**
- **UC07 - Hyperparameter kalibrieren**
- **UC08 - Validierungskurven visualisieren**
- **UC09 - Modelle trainieren**
- **UC10 - Trainingszeiten berechnen**
- **UC11 - Trainingszeiten visualisieren**
- **UC12 - Entscheidungsbaum visualisieren**

Zunächst wird definiert, welcher Wert vorhergesagt werden soll und welche Prädikatoren dafür verwendet werden.

```
In [7]: # define label --> target value, which should be predicted
        y = df["Duration"]
        # define features/ variables for predicting the label
        X = df.drop(["Duration"], axis = 1)
```

Im nachfolgendem Schritt wird ein Testdesign erstellt. Dafür werden die Daten in Trainings- und Testdaten aufgeteilt. Zum Testen der Vorhersagequalität werden die Modelle mit den Trainingsdaten trainiert und mit den Testdaten getestet. Um repräsentativere Ergebnisse über die Qualität zu erhalten, wird eine 10-fold cross validation durchgeführt.

```
In [8]: # create a test design
        from sklearn.model_selection import train_test_split

        # split data into train and test set
        X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

        from sklearn.model_selection import KFold
        from sklearn.model_selection import cross_val_score

        # 10-fold cross validation to get more representative test results
        kf = KFold(n_splits = 10)
```

Es werden vier Regressionsalgorithmen zum Lernen der Beziehungen zwischen den Daten verwendet:

- Lineare Regression

- Bayesian Regression
- Support Vector Regression
- Decision Tree Regression

```
In [9]: # load algorithms from scikit learn libary
        from sklearn.linear_model import LinearRegression
        from sklearn.linear_model import BayesianRidge
        from sklearn.svm import SVR
        from sklearn.tree import DecisionTreeRegressor

        # initialize models
        lr = LinearRegression()
        br = BayesianRidge()
        svr = SVR()
        dtr =  DecisionTreeRegressor()

        # define model names
        model_names = (
            'Lineare Regression',
            'Bayesian Regression',
            'Support Vector Regression',
            'Decision Tree Regression'
        )

        # function to get the name of a model for printing
        # @param model_var: model for which the name is searched
        # @return switcher.get(): name of the model
        def get_model_name(model_var):
            switcher = {
                lr:model_names[0],
                br:model_names[1],
                svr:model_names[2],
                dtr:model_names[3]
            }
            return switcher.get(model_var, "Invalide model")
```

Für die Support Vector Regression und die Decision Tree Regression werden die Hyperparameter auf optimale Werte kalibriert. Für die anderen beiden Methoden entfällt die Optimierung aufgrund der geringeren Komplexität der Algorithmen. Das Ändern der Parameter würde sich im Ergebnis kaum bis gar nicht bemerkbar machen.
Damit die Findung eines optimalen Wertes für einen Hyperparameter nachvollzogen werden kann, wird die folgende Funktion verwendet, um die validation curve zu visualisieren.

```
In [10]: from sklearn.model_selection import validation_curve
         import matplotlib.pyplot as plt

         # function to visualize the validation curve for hyperparameters
         # @param model: model with the hyperparameter
```

```
        # @param param: hyperparameter which should be optimized
        # @param param_range: values for the hyperparameter
        def draw_val_curve(model, param, param_range):
            # calculate validation curves
            train_scores, test_scores = validation_curve(
                model,
                X,
                y,
                param_name = param,
                param_range = param_range
            )

            # print train score for each parameter value in the range
            print("Train Scores: " + str(np.mean(train_scores, axis = 1)))
            # print test score for each parameter value in the range
            print("Test Scores: " + str(np.mean(test_scores, axis = 1)))

            # set title of the diagram
            plt.title(param + '-Validierung')
            # label y axis
            plt.ylabel('Genauigkeit')
            # label x axis
            plt.xlabel('Parameter Werte')

            # draw curve of average training scores for each value in the range
            plt.plot(param_range, np.mean(train_scores, axis = 1), color = 'black',
               label = 'Training Data')
            # draw curve of average test scores for each value in the range
            plt.plot(param_range, np.mean(test_scores, axis = 1), linestyle='dashed',
               color = 'black', label = 'Test Data')
            # draw a legend for the both curves
            plt.legend()
            # show graph
            plt.show()
```

Mit der nächsten Funktion wird eine grid search zur empirischen Findung der optimalen Werte für die Hyperparameter der Support Vector Regression implementiert.

```
In [11]: from sklearn.pipeline import Pipeline
         from sklearn.preprocessing import StandardScaler
         from sklearn.model_selection import GridSearchCV

         # function to optimize the hyperparameters of a Support Vector Regression model
         # using a GridSearch to find the best values
         # @param svr_model: Support Vector Regression model which should be optimized
         def find_best_svr_params(svr_model):
             # create a pipeline with standardized features and the model
             pipeline = Pipeline([
```

```
        ("scaler", StandardScaler()),
        ("svr", svr_model)
    ])

    # GridSearch for the pipeline with the hyperparameters and the values for them
    gsCV = GridSearchCV(pipeline, param_grid = {
        "svr__C": [0.1, 0.5, 1.0, 1.5, 2.0, 2.5],
        "svr__epsilon": [0.0, 0.5, 1.0, 1.5, 2.0],
        "svr__kernel": ['rbf','linear']
    })

    # fit the data to the GridSearch as a model
    gsCV.fit(X,y)

    # print best values for the hyperparameters
    print("Bester Wert für SVR-Hyperparameter: " + str(gsCV.best_params_))

    # set best values as parameters for the Support Vector Regression model
    svr_model.C = gsCV.best_params_["svr__C"]
    svr_model.epsilon = gsCV.best_params_["svr__epsilon"]
    svr_model.kernel = gsCV.best_params_["svr__kernel"]
```

```
In [12]: # call function to find the best values for the hyperparameters
         find_best_svr_params(svr)
```

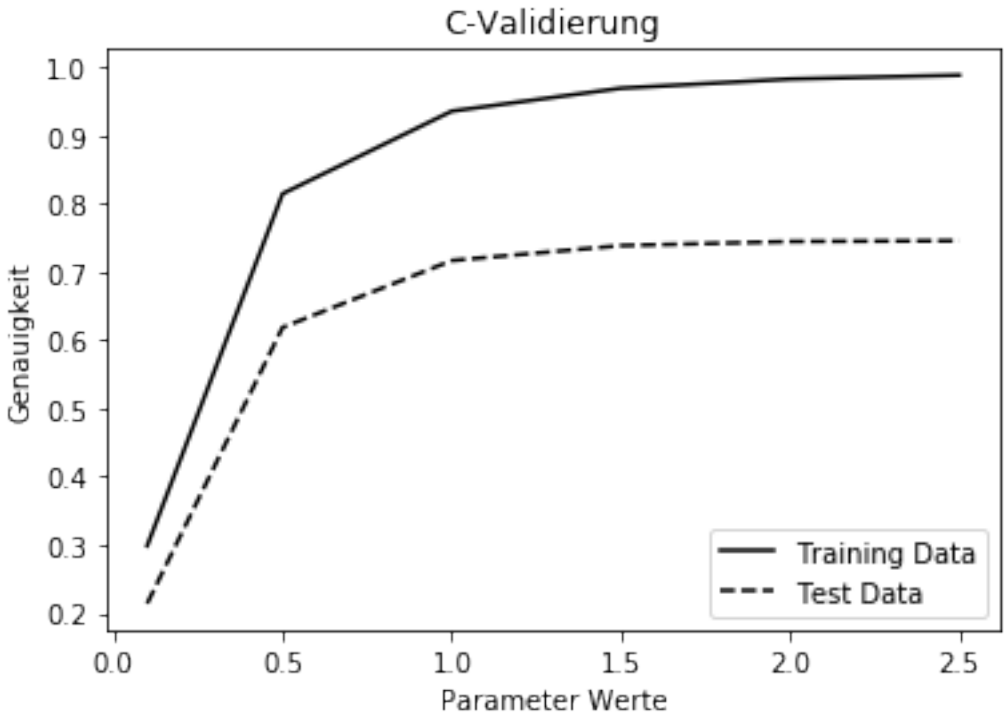
```
Bester Wert für SVR-Hyperparameter:
 {'svr__C': 2.5, 'svr__epsilon': 0.5, 'svr__kernel': 'linear'}
```

Für den C, epsilon und kernel Parameter wird eine validation curve erstellt.

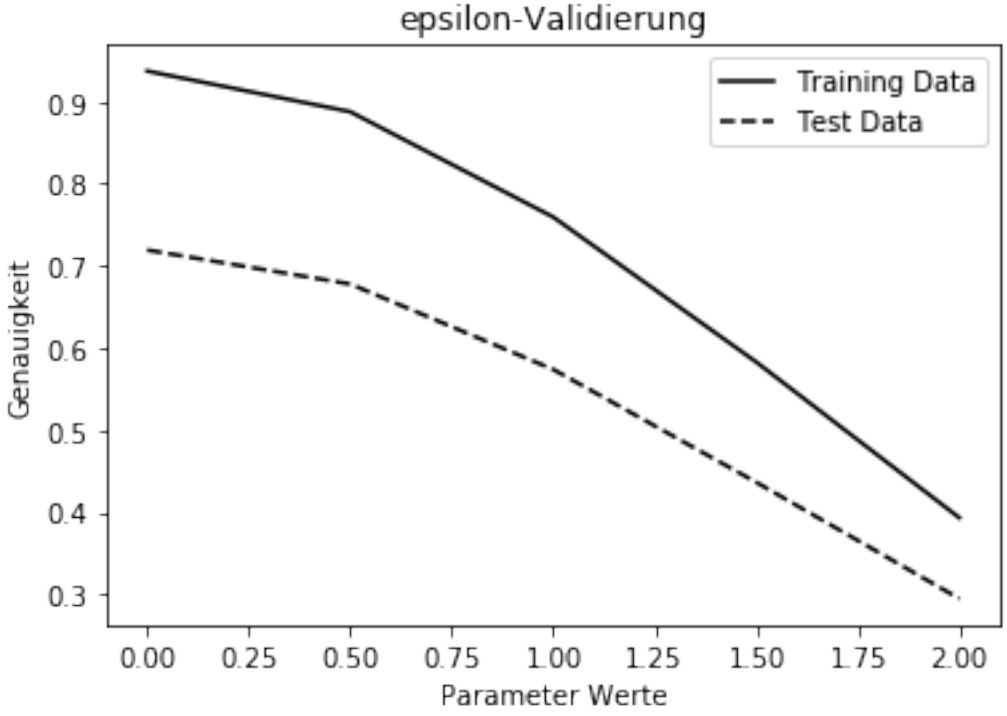
Je geringer die Genauigkeitswerte auf der y-Achse sind, desto höher ist der Bias. Wenn der Bias hoch ist, konnten die Zusammenhänge in den Daten nicht erkannt werden, weshalb die Vorhersageleistung abnimmt. In diesem Fall ist von einer Unteranpassung des Modells zu sprechen. Je weiter die Kurven für die Trainings- und Testergebnisse auseinander gehen, desto höher ist die Varianz. Das Modell ist in diesem Fall zu komplex geworden, weshalb es sich zu sehr an die Trainingsdaten angepasst hat.

```
In [13]: # call function to visualize the validation curve for the hyperparameters
         draw_val_curve(SVR(), "C", np.array([0.1, 0.5, 1.0, 1.5, 2.0, 2.5]))
         draw_val_curve(SVR(), "epsilon", np.array([0.0, 0.5, 1.0, 1.5, 2.0]))
         draw_val_curve(SVR(), "kernel", ['rbf', 'linear'])
```

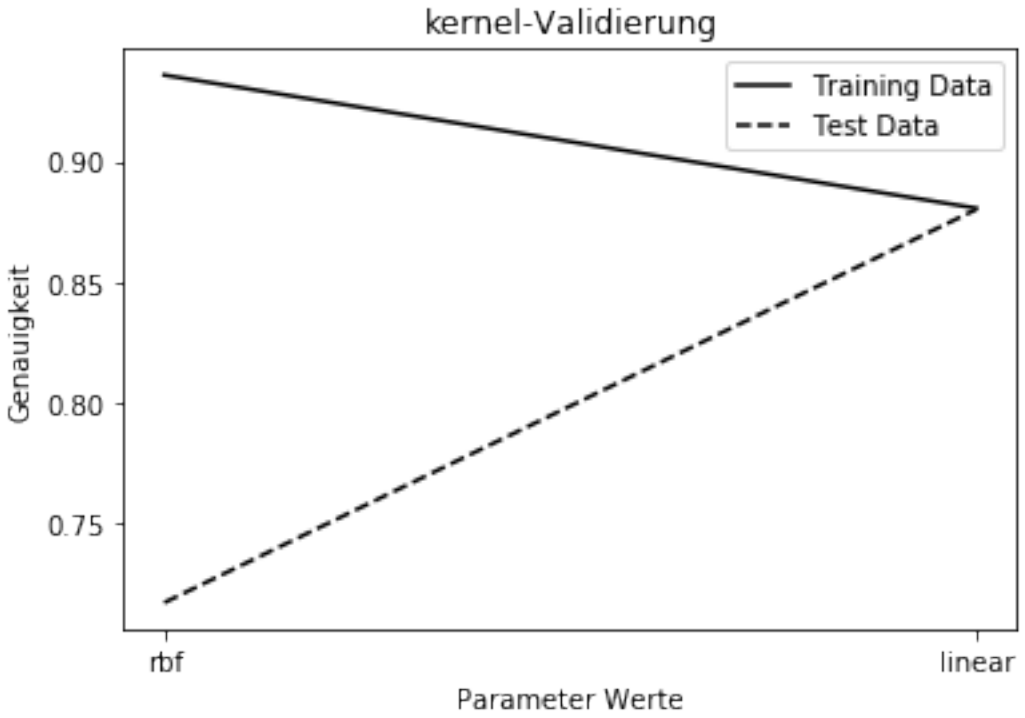
```
Train Scores: [0.29888079 0.81474769 0.93580508 0.9693997  0.98285859 0.98850692]
Test Scores: [0.21422706 0.61880717 0.71683665 0.73872624 0.74471136 0.74594193]
```




```
Train Scores: [0.93717043 0.88763834 0.75938633 0.58283479 0.39325996]
Test Scores: [0.71923343 0.67785072 0.57385061 0.43646789 0.29479446]
```



```
Train Scores: [0.93580508 0.8805365 ]
Test Scores: [0.71683665 0.88005764]
```



Für die Decision Tree Regression werden ebenfalls optimale Werte für die Hyperparameter per grid search gesucht.

```
In [14]: # function to optimize the hyperparameters of a Decision Tree Regression model
         # using a GridSearch to find the best values
         # @param dtr_model: Decision Tree Regression model which should be optimized
         def find_best_dtr_params(dtr_model):
             # create a pipeline with standardized features and the model
             pipeline = Pipeline([
                 ("scaler", StandardScaler()),
                 ("dtr", dtr)
             ])

             # GridSearch for the pipeline with the hyperparameters and the values for them
             gsCV = GridSearchCV(pipeline, param_grid = {
                 "dtr__max_depth": [1, 3, 5, 7, 9],
                 "dtr__max_leaf_nodes": [None, 2, 3, 4, 5, 6, 8, 9, 20],
                 "dtr__min_weight_fraction_leaf": [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

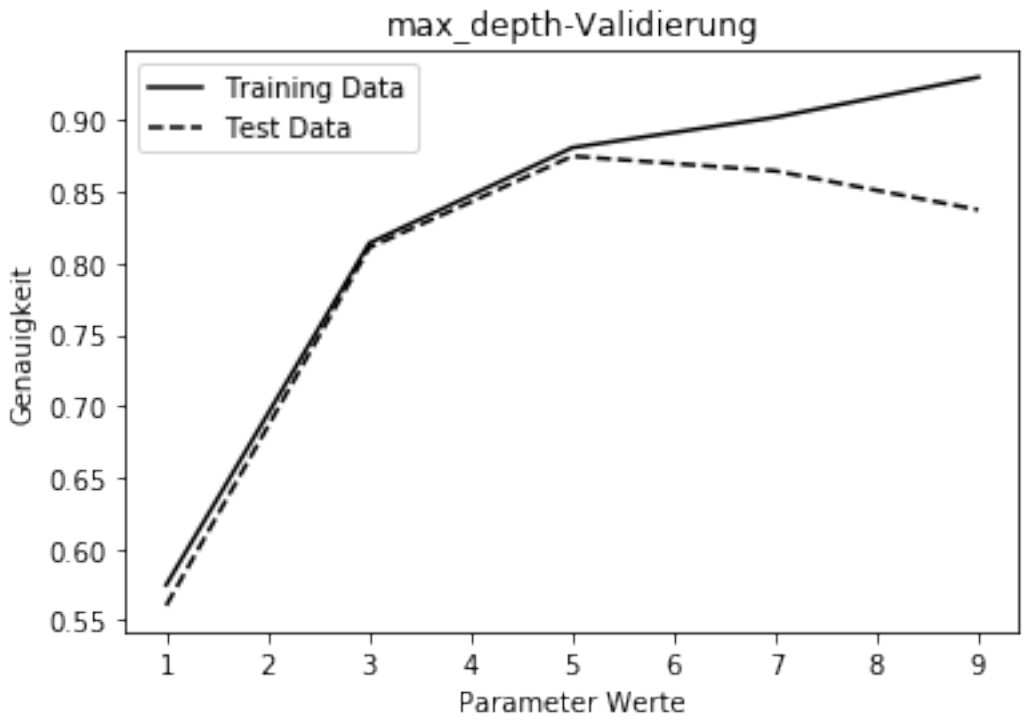
```
        })

        # fit the data to the GridSearch as a model
        gsCV.fit(X,y)

        # print best values for the hyperparameters
        print("Bester Wert für DTR-Hyperparameter: " + str(gsCV.best_params_))

        # set best values as parameters for the Decision Tree Regression model
        dtr_model.max_depth = gsCV.best_params_["dtr__max_depth"]
        dtr_model.max_leaf_nodes = gsCV.best_params_["dtr__max_leaf_nodes"]
        dtr_model.min_weight_fraction_leaf =
           gsCV.best_params_["dtr__min_weight_fraction_leaf"]
```

In [15]:
```
# call function to find the best values for the hyperparameters
find_best_dtr_params(dtr)
```

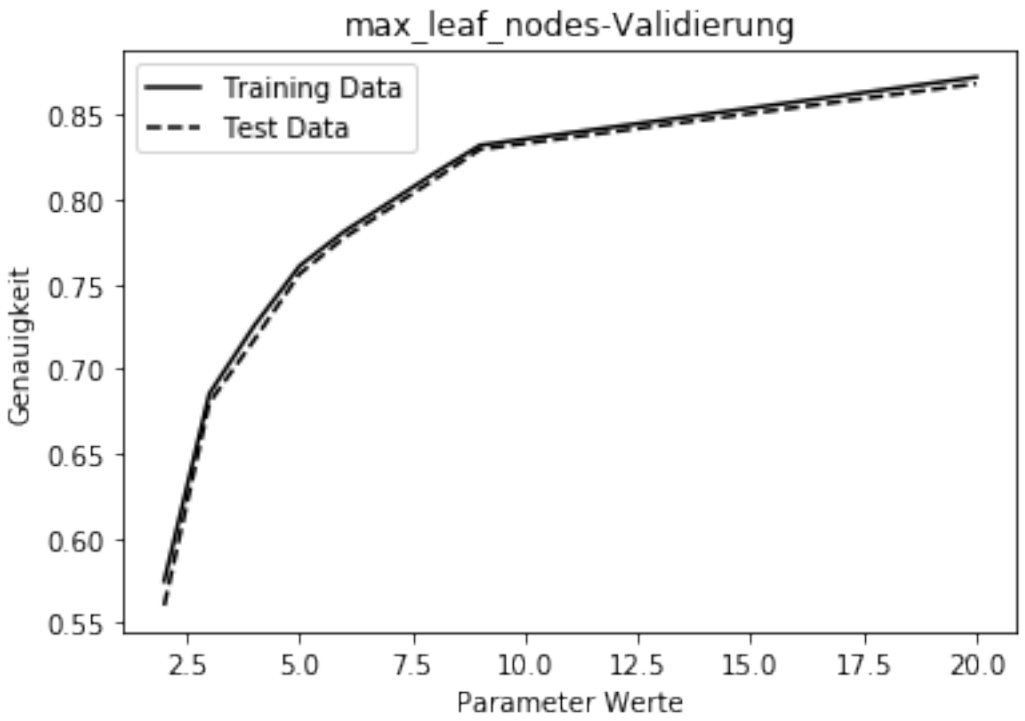
```
Bester Wert für DTR-Hyperparameter:
{'dtr__max_depth': 5, 'dtr__max_leaf_nodes': None, 'dtr__min_weight_fraction_leaf': 0.0}
```

In [16]:
```
# call function to visualize the validation curve for the hyperparameters
draw_val_curve(
       DecisionTreeRegressor(), "max_depth", np.array([1, 3, 5, 7, 9])
)

draw_val_curve(
       DecisionTreeRegressor(), "max_leaf_nodes", np.array([2, 3, 4, 5, 6, 8, 9, 20])
)

draw_val_curve(
       DecisionTreeRegressor(), "min_weight_fraction_leaf",
        np.array([0.0, 0.1, 0.2, 0.3, 0.4, 0.5])
)
```

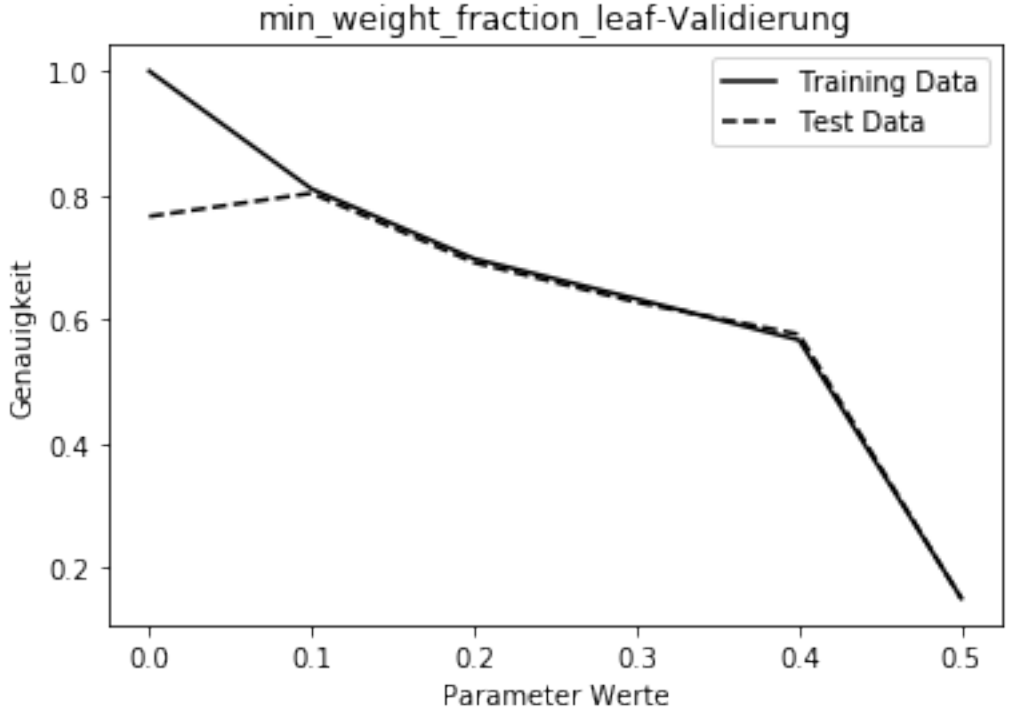
```
Train Scores: [0.57469735 0.81413222 0.88105713 0.90229685 0.93043298]
Test Scores: [0.56027153 0.81096092 0.87509956 0.86475797 0.83751204]
```




```
Train Scores: [0.57469735 0.68539106 0.72572382 0.76086184 0.78141575 0.81580126
 0.83214582 0.87235625]
Test Scores: [0.56027153 0.68003188 0.71739253 0.75619697 0.77778223 0.81209406
 0.82972405 0.86855594]
```




```
Train Scores: [1.         0.81011469 0.69772463 0.6330373  0.56676264 0.15077473]
Test Scores: [0.76595063 0.80329457 0.69172239 0.62771334 0.5760224  0.1502103 ]
```



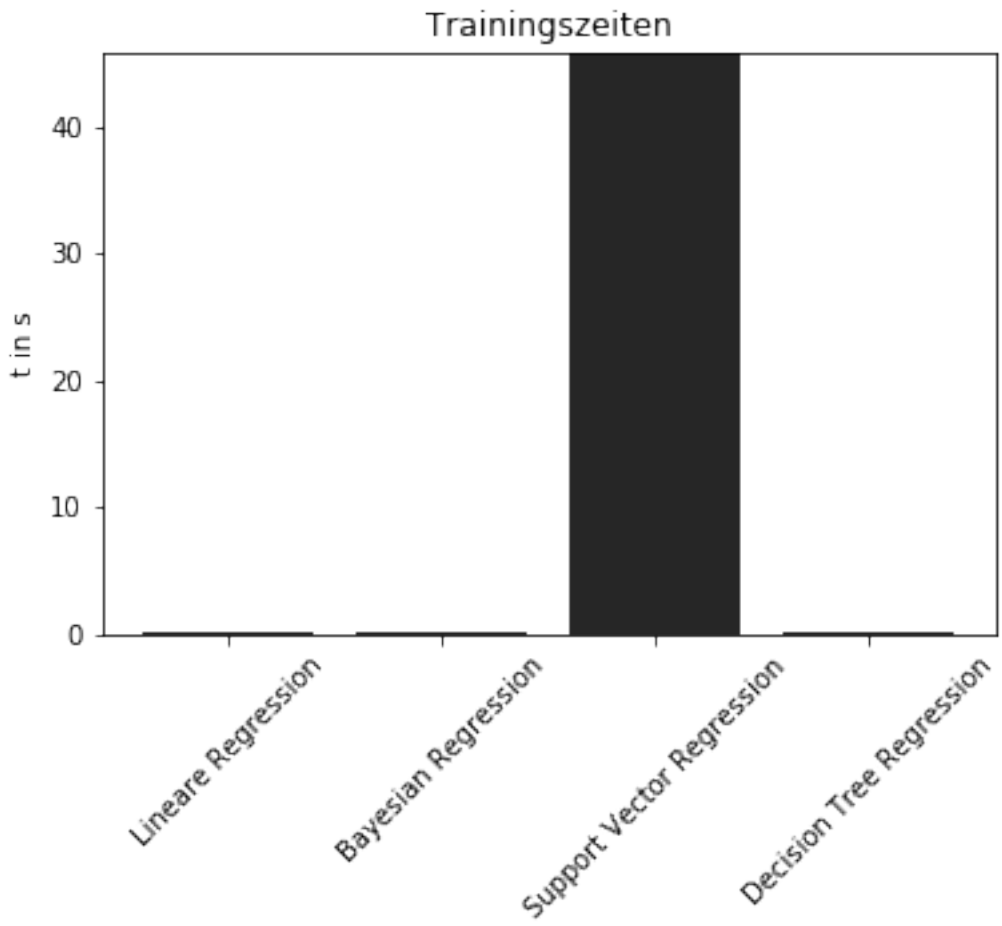
Die nächsten beiden Funktionen sind zur Anpassung der Modelle an die Trainingsdaten. Die Zeiten, die zum Trainieren benötigt werden, werden dabei gemessen und vergleichend in einem Balkendiagramm dargestellt.

In [17]:
```
from timeit import default_timer as timer

        # function to calculate the time which a model needs to fit the data
        # @param model: model which should be trained
        # @return tt: training time
        def calculate_training_time(model):
            # start the measurement of the training time
            start = timer()

            # train the model with the test data
            model.fit(X_train, y_train)

            # stop the measurement of the training time
            end = timer()

            # calculate the time from start to end
            tt = end - start

            # print the training time of the model
            print("Trainingszeit der " + get_model_name(model) + ": " + str(tt))

            # return the training time
            return tt

        # function to draw a  bar graph of the training times
        # @param values: calculated training times of all models
        def draw_training_times(values):
            # create a figure
            fig,ax = plt.subplots()

            # set title
            plt.title('Trainingszeiten')
            # label y axis
            plt.ylabel('t in s')

            # set limits for the y axis
            y_pos = np.arange(len(model_names))
            plt.ylim(0,max(values))

            # draw bars
            plt.bar(y_pos, values, align='center', alpha=0.85, color='black')
```

```
            # print names of the models
            ax.set_xticks(range(len(model_names)))
            ax.set_xticklabels(model_names, rotation='45')

            # show graph
            plt.show()

            #plt.savefig('election_presidentielle.png')
```

In [18]:
```
# fit models to the train data and calculate training times
lr_tt = calculate_training_time(lr)
br_tt = calculate_training_time(br)
svr_tt = calculate_training_time(svr)
dtr_tt = calculate_training_time(dtr)

# call function to draw the graph of the training times
draw_training_times([lr_tt, br_tt, svr_tt, dtr_tt])
```

```
Trainingszeit der Lineare Regression: 0.0029520837067703496
Trainingszeit der Bayesian Regression: 0.006219597783209506
Trainingszeit der Support Vector Regression: 45.81353799254207
Trainingszeit der Decision Tree Regression: 0.0055067005368059085
```



Beim Entscheidungsbaum besteht der Vorteil, dass er auf einer Struktur und auf keiner Anpassungsfunktion basiert. Somit kann eine Vorhersage besser nachvollzogen werden. Aus diesem Grund wird der generierte Entscheidungsbaum textuell und im Anschluss grafisch ausgegeben.

```
In [19]: from sklearn.tree import export_graphviz

         # get a viewable structur of the generated decision tree
         tree = export_graphviz(dtr, None, feature_names = list(X), rounded = True)

         # print decision tree in textual form
         print(tree)
```

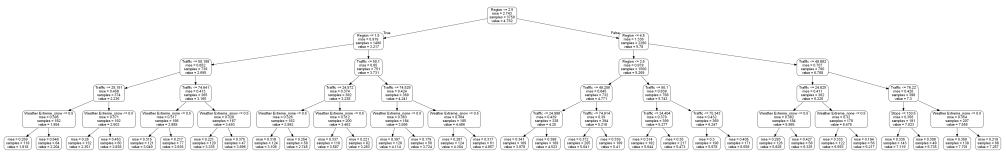
```
digraph Tree {
node [shape=box, style="rounded", color="black", fontname=helvetica] ;
edge [fontname=helvetica] ;
0 [label="Region <= 2.5\nmse = 2.743\nsamples = 3750\nvalue = 4.762"] ;
1 [label="Region <= 1.5\nmse = 0.919\nsamples = 1490\nvalue = 3.217"] ;
0 -> 1 [labeldistance=2.5, labelangle=45, headlabel="True"] ;
2 [label="Traffic <= 50.198\nmse = 0.652\nsamples = 739\nvalue = 2.695"] ;
```

```
1 -> 2 ;
3 [label="Traffic <= 25.151\nmse = 0.458\nsamples = 374\nvalue = 2.236"] ;
2 -> 3 ;
4 [label="Weather Extreme_snow <= 0.5\nmse = 0.395\nsamples = 182\nvalue = 1.954"] ;
3 -> 4 ;
5 [label="mse = 0.259\nsamples = 118\nvalue = 1.818"] ;
4 -> 5 ;
6 [label="mse = 0.548\nsamples = 64\nvalue = 2.204"] ;
4 -> 6 ;
7 [label="Weather Extreme_snow <= 0.5\nmse = 0.371\nsamples = 192\nvalue = 2.503"] ;
3 -> 7 ;
8 [label="mse = 0.26\nsamples = 132\nvalue = 2.351"] ;
7 -> 8 ;
9 [label="mse = 0.453\nsamples = 60\nvalue = 2.838"] ;
7 -> 9 ;
10 [label="Traffic <= 74.641\nmse = 0.413\nsamples = 365\nvalue = 3.165"] ;
2 -> 10 ;
11 [label="Weather Extreme_none <= 0.5\nmse = 0.317\nsamples = 198\nvalue = 2.889"] ;
10 -> 11 ;
12 [label="mse = 0.315\nsamples = 121\nvalue = 3.049"] ;
11 -> 12 ;
13 [label="mse = 0.217\nsamples = 77\nvalue = 2.636"] ;
11 -> 13 ;
14 [label="Weather Extreme_snow <= 0.5\nmse = 0.328\nsamples = 167\nvalue = 3.493"] ;
10 -> 14 ;
15 [label="mse = 0.221\nsamples = 120\nvalue = 3.335"] ;
14 -> 15 ;
16 [label="mse = 0.375\nsamples = 47\nvalue = 3.896"] ;
14 -> 16 ;
17 [label="Traffic <= 50.1\nmse = 0.65\nsamples = 751\nvalue = 3.731"] ;
1 -> 17 ;
18 [label="Traffic <= 24.572\nmse = 0.374\nsamples = 382\nvalue = 3.238"] ;
17 -> 18 ;
19 [label="Weather Extreme_none <= 0.5\nmse = 0.326\nsamples = 182\nvalue = 2.992"] ;
18 -> 19 ;
20 [label="mse = 0.318\nsamples = 124\nvalue = 3.108"] ;
19 -> 20 ;
21 [label="mse = 0.254\nsamples = 58\nvalue = 2.745"] ;
19 -> 21 ;
22 [label="Weather Extreme_none <= 0.5\nmse = 0.312\nsamples = 200\nvalue = 3.462"] ;
18 -> 22 ;
23 [label="mse = 0.337\nsamples = 118\nvalue = 3.587"] ;
22 -> 23 ;
24 [label="mse = 0.221\nsamples = 82\nvalue = 3.283"] ;
22 -> 24 ;
25 [label="Traffic <= 74.528\nmse = 0.424\nsamples = 369\nvalue = 4.241"] ;
17 -> 25 ;
26 [label="Weather Extreme_none <= 0.5\nmse = 0.363\nsamples = 184\nvalue = 3.995"] ;
```

```
25 -> 26 ;
27 [label="mse = 0.397\nsamples = 128\nvalue = 4.114"] ;
26 -> 27 ;
28 [label="mse = 0.179\nsamples = 56\nvalue = 3.724"] ;
26 -> 28 ;
29 [label="Weather Extreme_snow <= 0.5\nmse = 0.364\nsamples = 185\nvalue = 4.486"] ;
25 -> 29 ;
30 [label="mse = 0.287\nsamples = 124\nvalue = 4.304"] ;
29 -> 30 ;
31 [label="mse = 0.317\nsamples = 61\nvalue = 4.857"] ;
29 -> 31 ;
32 [label="Region <= 4.5\nmse = 1.336\nsamples = 2260\nvalue = 5.78"] ;
0 -> 32 [labeldistance=2.5, labelangle=-45, headlabel="False"] ;
33 [label="Region <= 3.5\nmse = 0.878\nsamples = 1500\nvalue = 5.269"] ;
32 -> 33 ;
34 [label="Traffic <= 49.206\nmse = 0.645\nsamples = 732\nvalue = 4.771"] ;
33 -> 34 ;
35 [label="Traffic <= 24.988\nmse = 0.439\nsamples = 338\nvalue = 4.25"] ;
34 -> 35 ;
36 [label="mse = 0.341\nsamples = 169\nvalue = 3.978"] ;
35 -> 36 ;
37 [label="mse = 0.388\nsamples = 169\nvalue = 4.523"] ;
35 -> 37 ;
38 [label="Traffic <= 74.914\nmse = 0.39\nsamples = 394\nvalue = 5.218"] ;
34 -> 38 ;
39 [label="mse = 0.372\nsamples = 205\nvalue = 5.041"] ;
38 -> 39 ;
40 [label="mse = 0.339\nsamples = 189\nvalue = 5.41"] ;
38 -> 40 ;
41 [label="Traffic <= 50.1\nmse = 0.639\nsamples = 768\nvalue = 5.743"] ;
33 -> 41 ;
42 [label="Traffic <= 24.404\nmse = 0.379\nsamples = 399\nvalue = 5.277"] ;
41 -> 42 ;
43 [label="mse = 0.314\nsamples = 182\nvalue = 5.044"] ;
42 -> 43 ;
44 [label="mse = 0.35\nsamples = 217\nvalue = 5.473"] ;
42 -> 44 ;
45 [label="Traffic <= 75.343\nmse = 0.432\nsamples = 369\nvalue = 6.247"] ;
41 -> 45 ;
46 [label="mse = 0.3\nsamples = 198\nvalue = 5.978"] ;
45 -> 46 ;
47 [label="mse = 0.405\nsamples = 171\nvalue = 6.558"] ;
45 -> 47 ;
48 [label="Traffic <= 49.882\nmse = 0.707\nsamples = 760\nvalue = 6.788"] ;
32 -> 48 ;
49 [label="Traffic <= 24.825\nmse = 0.411\nsamples = 362\nvalue = 6.226"] ;
48 -> 49 ;
50 [label="Weather Extreme_snow <= 0.5\nmse = 0.382\nsamples = 184\nvalue = 5.985"] ;
```

```
49 -> 50 ;
51 [label="mse = 0.283\nsamples = 126\nvalue = 5.828"] ;
50 -> 51 ;
52 [label="mse = 0.427\nsamples = 58\nvalue = 6.325"] ;
50 -> 52 ;
53 [label="Weather Extreme_none <= 0.5\nmse = 0.32\nsamples = 178\nvalue = 6.475"] ;
49 -> 53 ;
54 [label="mse = 0.333\nsamples = 122\nvalue = 6.593"] ;
53 -> 54 ;
55 [label="mse = 0.194\nsamples = 56\nvalue = 6.217"] ;
53 -> 55 ;
56 [label="Traffic <= 76.22\nmse = 0.426\nsamples = 398\nvalue = 7.3"] ;
48 -> 56 ;
57 [label="Stops <= 130.5\nmse = 0.356\nsamples = 191\nvalue = 7.023"] ;
56 -> 57 ;
58 [label="mse = 0.335\nsamples = 143\nvalue = 7.119"] ;
57 -> 58 ;
59 [label="mse = 0.308\nsamples = 48\nvalue = 6.735"] ;
57 -> 59 ;
60 [label="Weather Extreme_none <= 0.5\nmse = 0.354\nsamples = 207\nvalue = 7.555"] ;
56 -> 60 ;
61 [label="mse = 0.356\nsamples = 139\nvalue = 7.701"] ;
60 -> 61 ;
62 [label="mse = 0.219\nsamples = 68\nvalue = 7.257"] ;
60 -> 62 ;
}
```

```
In [20]: import graphviz

         # visualize the decision tree in a tree
          graphviz.Source(tree)

         # for saving the decision tree as png use the following code
         #src = graphviz.Source(tree, format = "png")
         #src.render("./dtr")
```



Evaluation

- **UC13 - Kennzahlen berechnen**
- **UC14 - Ergebnisse der Kennzahlen visualisieren**
- **UC15 - Lernkurven visualisieren**
- **UC16 - Lieferdauer vorhersagen**

Zur Evaluierung werden im ersten Schritt Metriken zur Qualitätsmessung implementiert. Dabei werden die Kennzahlen R2 (Bestimmtheitsmaß), mean squared error, root mean squared error und mean absolute error berechnet. Die Ergebnisse werden danach für die Algorithmen vergleichend in einem Balkendiagramm dargestellt.

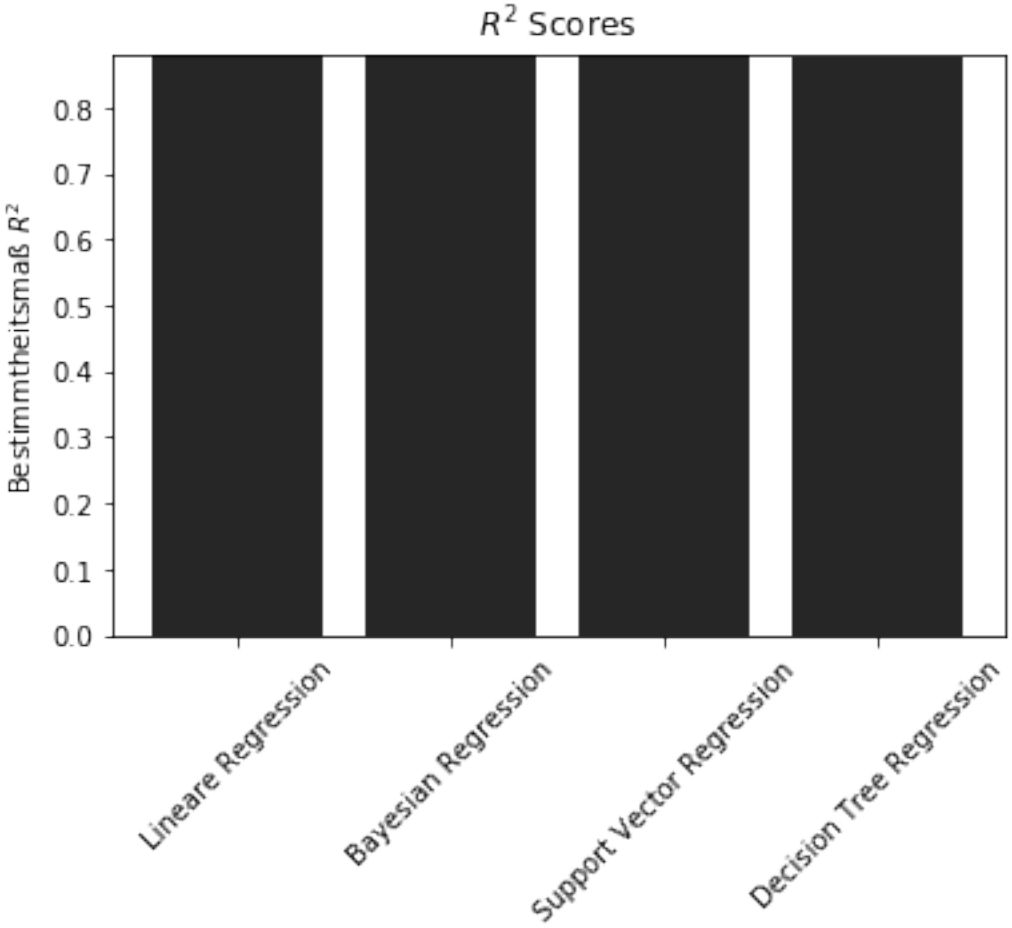
```
In [21]: # function to calculate quality scores
         # @param model: model which should be tested
         # @param metric: metric to test the model
         # @return score: calculated metric score
         def evaluate_model(model, metric):
             # print model name
             print(get_model_name(model) + ": ")

             # check if the metric is mse, rmse, mae
             # these metrics returns negative values
             # more infos on: https://github.com/scikit-learn/scikit-learn/issues/2439
             if metric.find("neg") == 0:
                 # calculate metric score and use 10-fold cross validation
                 # reverse sign
                 scores = -cross_val_score(model, X, y, cv = kf, scoring = metric)
             else:
                 # calculate metric score and use 10-fold cross validation
                 scores = cross_val_score(model, X, y, cv = kf, scoring = metric)

             # print the scores from the 10-fold cross validation
             print(scores)

             # print average score
             score = np.mean(scores)
             print("Durchschnittswert: " + str(score) + '\n')

             # return the calculated score
             return score

         # function to draw a  bar graph of the evaluation results
         # @param title: title of the graph
         # @param ylabel: label for the y axis
         # @param values: calculated evaluation results
         def draw_metric_scores(title, ylabel, values):
             # create a figure
             fig,ax = plt.subplots()

             # set title
             plt.title(title)
             # label y axis
             plt.ylabel(ylabel)

             # set limits for the y axis
```

```
    y_pos = np.arange(len(model_names))

    # R2 score has another range
    if title == '$R^2$':
        plt.ylim(0, 1)
        #plt.ylim(min(values) - 0.01, 1)
    else:
        plt.ylim(0,max(values))

    # draw bars
    plt.bar(y_pos, values, align='center', alpha=0.85, color='black')

    # print names of the models
    ax.set_xticks(range(len(model_names)))
    ax.set_xticklabels(model_names, rotation='45')

    # show graph
    plt.show()
```

```
In [22]: # calculate R2 scores for each model
         lr_r2 = evaluate_model(lr,'r2')
         br_r2 = evaluate_model(br,'r2')
         svr_r2 = evaluate_model(svr,'r2')
         dtr_r2 = evaluate_model(dtr,'r2')

         # call function to draw the graph of the R^2 scores
         draw_metric_scores(
                '$R^2$ Scores', 'Bestimmtheitsmaß $R^2$', [lr_r2, br_r2, svr_r2, dtr_r2]
         )
```

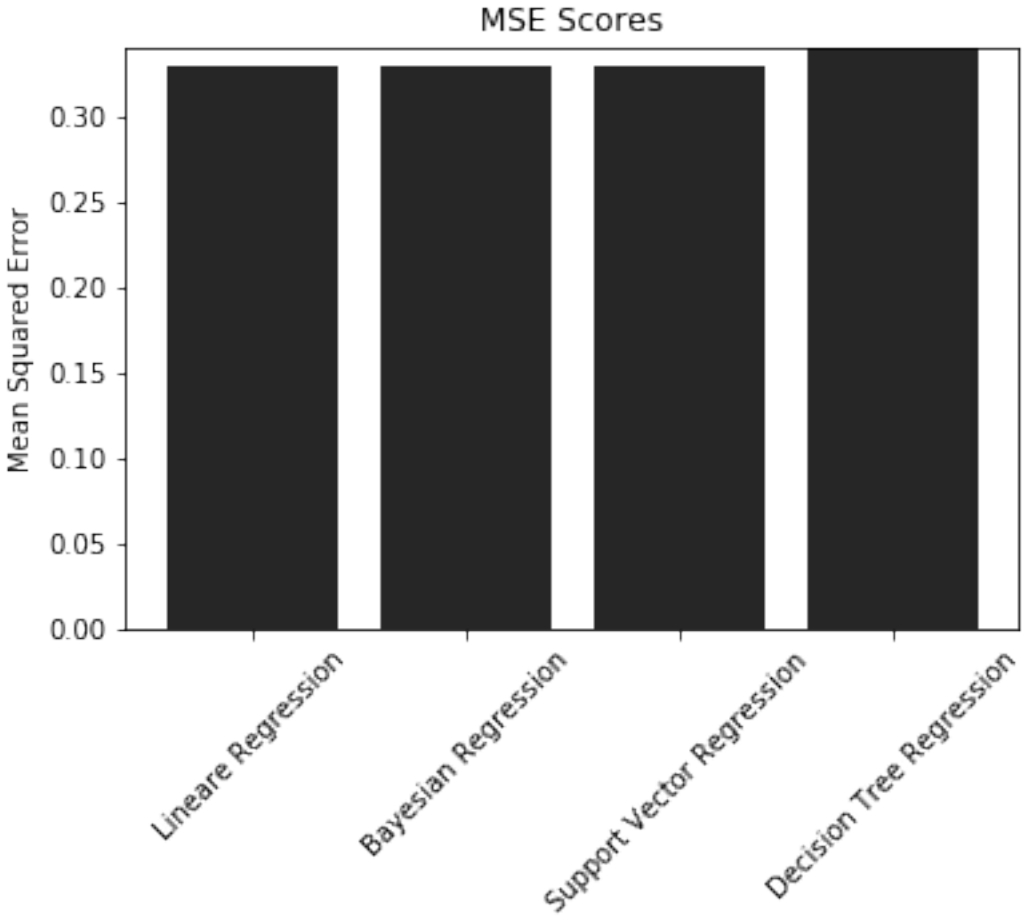
```
Lineare Regression:
[0.87795861 0.87082261 0.87498539 0.87654339 0.87783813 0.88868104
 0.88212928 0.88378808 0.88710123 0.88097002]
Durchschnittswert: 0.8800817780166457

Bayesian Regression:
[0.87797123 0.87079174 0.8749844  0.87656568 0.87784822 0.88867149
 0.88216331 0.88380428 0.88707612 0.88093918]
Durchschnittswert: 0.8800815657723546

Support Vector Regression:
[0.87800177 0.87040586 0.87502528 0.87628202 0.87787608 0.88896472
 0.88225714 0.88378651 0.88728618 0.880758  ]
Durchschnittswert: 0.8800643550688996

Decision Tree Regression:
[0.87311745 0.86954007 0.86592385 0.86894264 0.87081951 0.88913215
```

```
 0.88160961 0.88319005 0.88048576 0.87917242]
Durchschnittswert: 0.8761933518523801
```

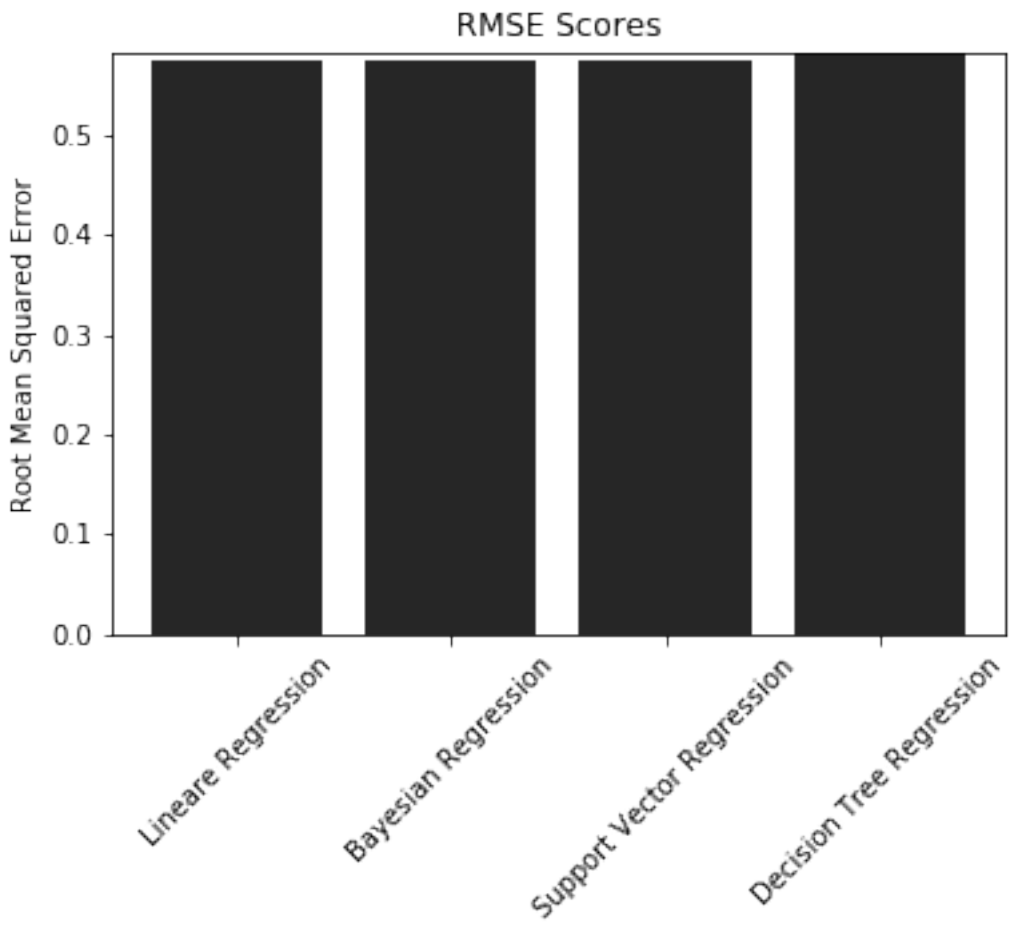



```
In [23]: from sklearn.metrics import mean_squared_error

         # calculate MSE for each model
         lr_mse = evaluate_model(lr,'neg_mean_squared_error')
         br_mse = evaluate_model(br,'neg_mean_squared_error')
         svr_mse = evaluate_model(svr,'neg_mean_squared_error')
         dtr_mse = evaluate_model(dtr,'neg_mean_squared_error')

         # call function to draw MSEs
         draw_metric_scores(
                'MSE Scores', 'Mean Squared Error', [lr_mse, br_mse, svr_mse, dtr_mse]br
         )
```

```
Lineare Regression:
[0.34520732 0.35672566 0.35325251 0.3033616  0.33173735 0.31432621
 0.3261642  0.32491924 0.31525353 0.32393514]
Durchschnittswert: 0.32948827619298116

Bayesian Regression:
[0.3451716  0.3568109  0.3532553  0.30330682 0.33170995 0.31435319
 0.32607002 0.32487395 0.31532367 0.32401907]
Durchschnittswert: 0.329489447099134

Support Vector Regression:
[0.34508523 0.35787651 0.3531398  0.30400384 0.3316343  0.3135252
 0.32581038 0.32492364 0.3147371  0.32451216]
Durchschnittswert: 0.3295248170146045

Decision Tree Regression:
[0.35890104 0.3602674  0.37885762 0.32203841 0.35079681 0.31305243
 0.3276022  0.32659128 0.33372628 0.32882724]
Durchschnittswert: 0.3400660708305701
```

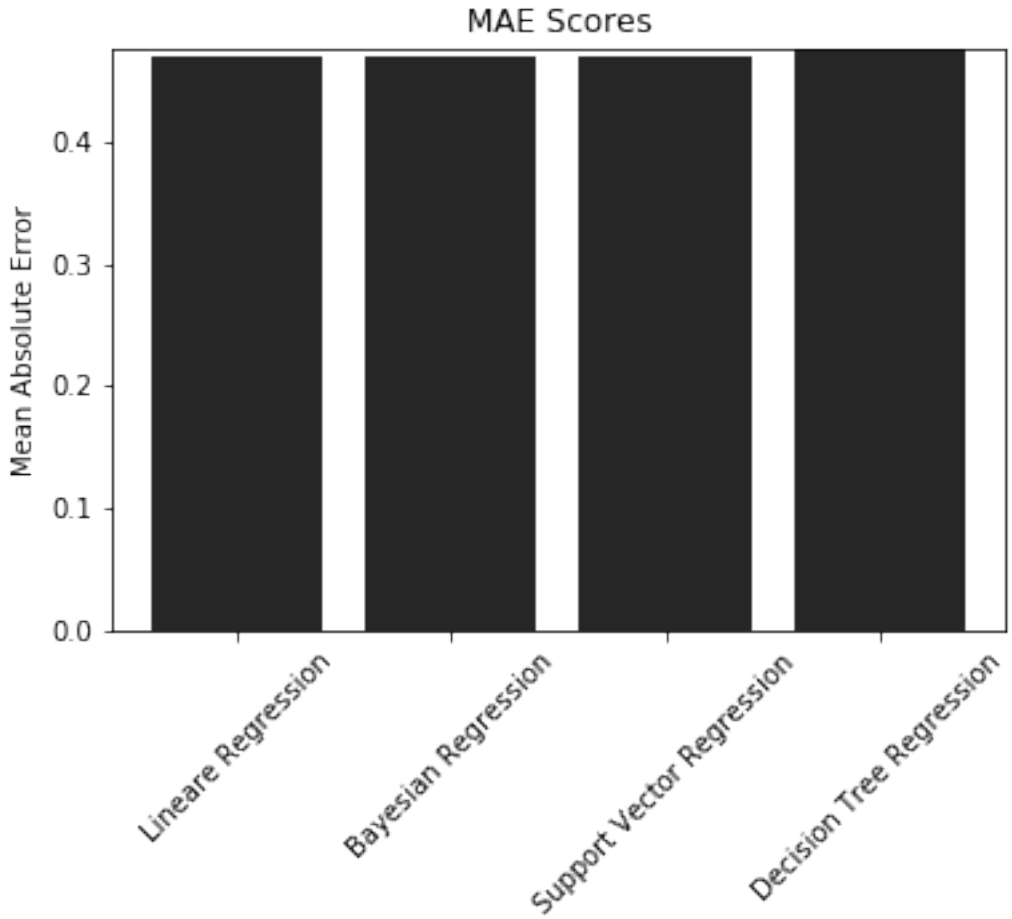


```
In [24]: from math import sqrt

         # calculate RMSE for each model

         lr_rmse = sqrt(lr_mse)
         print(model_names[0] + ": " + str(lr_rmse))

         br_rmse = sqrt(br_mse)
         print(model_names[1] + ": " + str(br_rmse))

         svr_rmse = sqrt(svr_mse)
         print(model_names[2] + ": " + str(svr_rmse))

         dtr_rmse = sqrt(dtr_mse)
         print(model_names[3] + ": " + str(dtr_rmse))

         # call function to draw RMSEs
         draw_metric_scores(
                'RMSE Scores', 'Root Mean Squared Error', [lr_rmse, br_rmse, svr_rmse, dtr_rmse]
         )

Lineare Regression: 0.5740106934482851
Bayesian Regression: 0.5740117133814727
Support Vector Regression: 0.5740425219568708
Decision Tree Regression: 0.5831518420022097
```




```
In [25]: from sklearn.metrics import mean_absolute_error

         # calculate MAE for each model
         lr_mae = evaluate_model(lr,'neg_mean_absolute_error')
         br_mae = evaluate_model(br,'neg_mean_absolute_error')
         svr_mae = evaluate_model(svr,'neg_mean_absolute_error')
         dtr_mae = evaluate_model(dtr,'neg_mean_absolute_error')

         # call function to draw MAEs
         draw_metric_scores(
                'MAE Scores', 'Mean Absolute Error', [lr_mae, br_mae, svr_mae, dtr_mae]
         )

Lineare Regression:
[0.47786181 0.48242529 0.49330886 0.44354722 0.47460802 0.45678307
 0.46982187 0.46926471 0.46344591 0.47249068]
Durchschnittswert: 0.47035574359490767
```

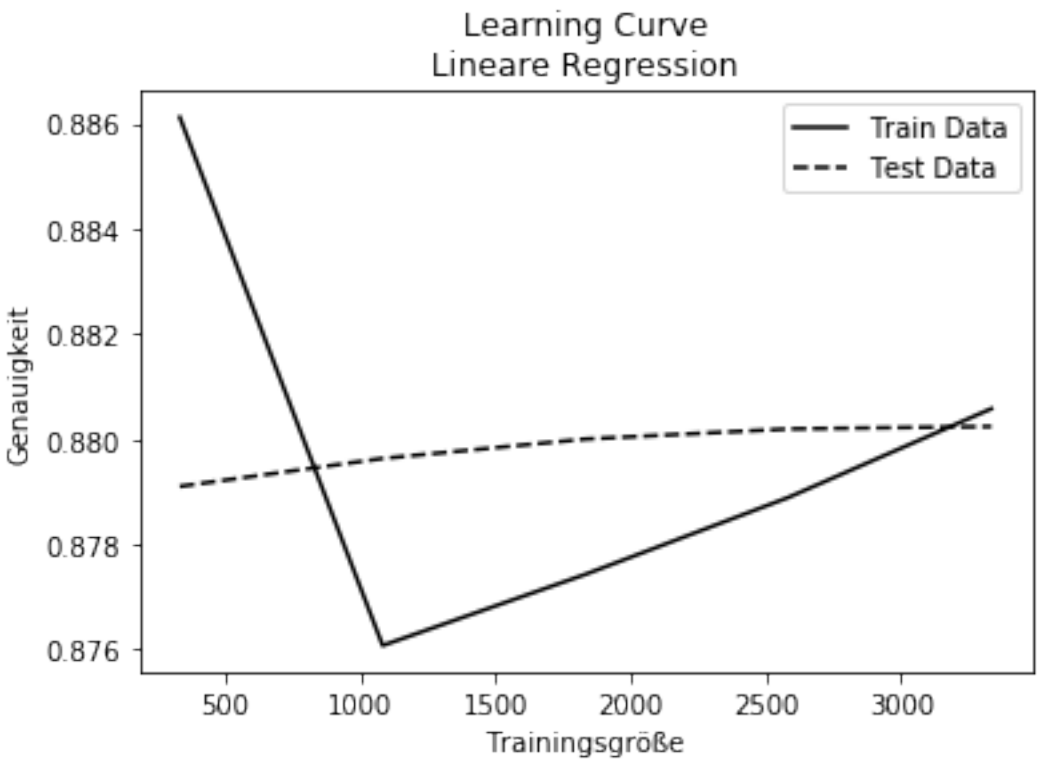
```
Bayesian Regression:
[0.47788278 0.48246118 0.49331617 0.44351759 0.47458964 0.4568135
 0.46968065 0.4692944  0.46354377 0.47254856]
Durchschnittswert: 0.47036482406734254

Support Vector Regression:
[0.47772384 0.48314421 0.4930058  0.4441468  0.4747677  0.45555087
 0.46951309 0.46892785 0.462723   0.47254279]
Durchschnittswert: 0.47020459425006045

Decision Tree Regression:
[0.49182565 0.49056163 0.50156561 0.46144784 0.4903602  0.44691778
 0.46617427 0.47143554 0.47367179 0.46980214]
Durchschnittswert: 0.47637624425177255
```

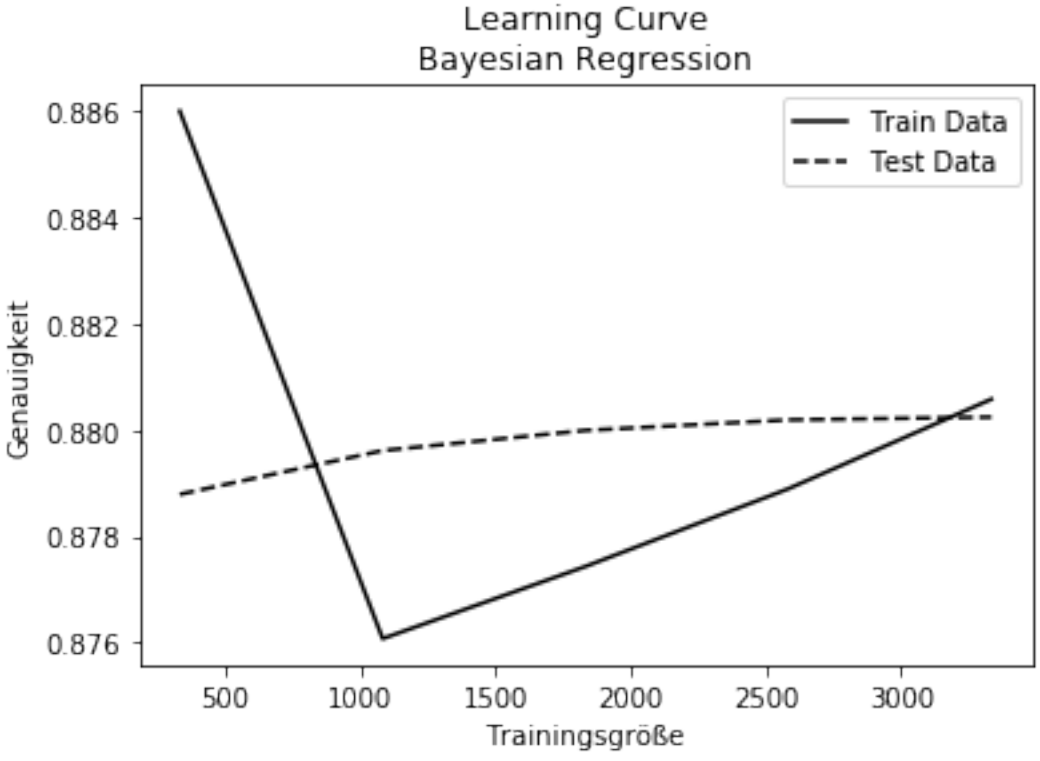


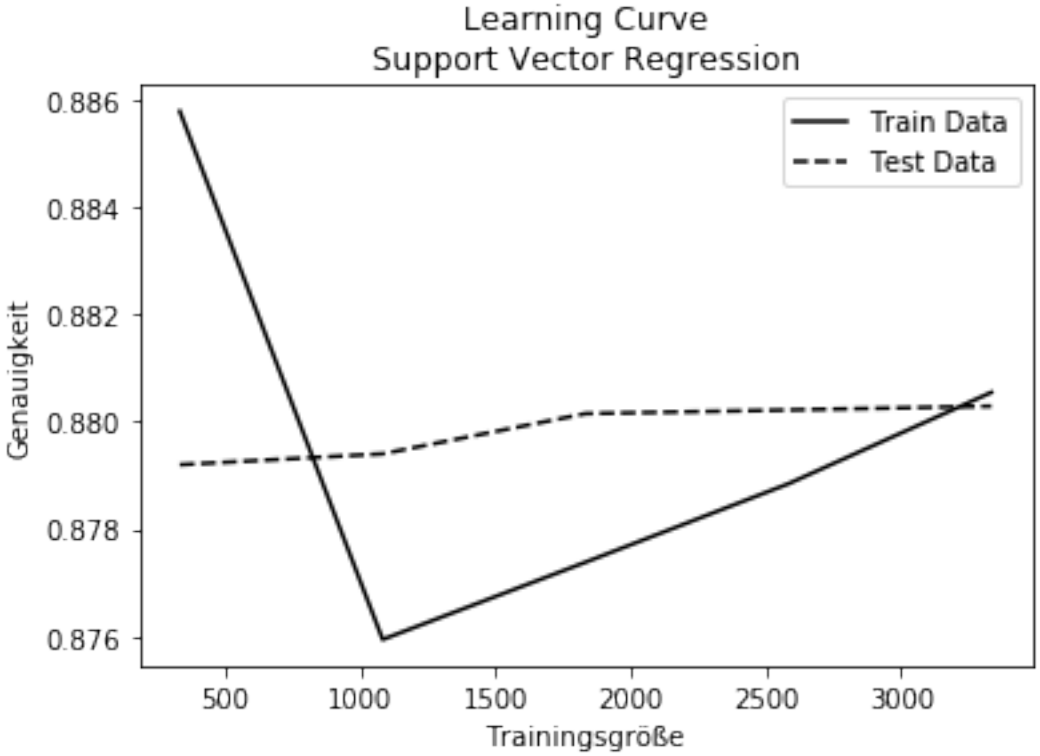
Im nächsten Schritt werden für alle Modelle die Lernkurven visualisiert. Durch die Darstellung lässt sich für jeden Algorithmus abschätzen, wie sich die Größe des Trainingssatzes auf die

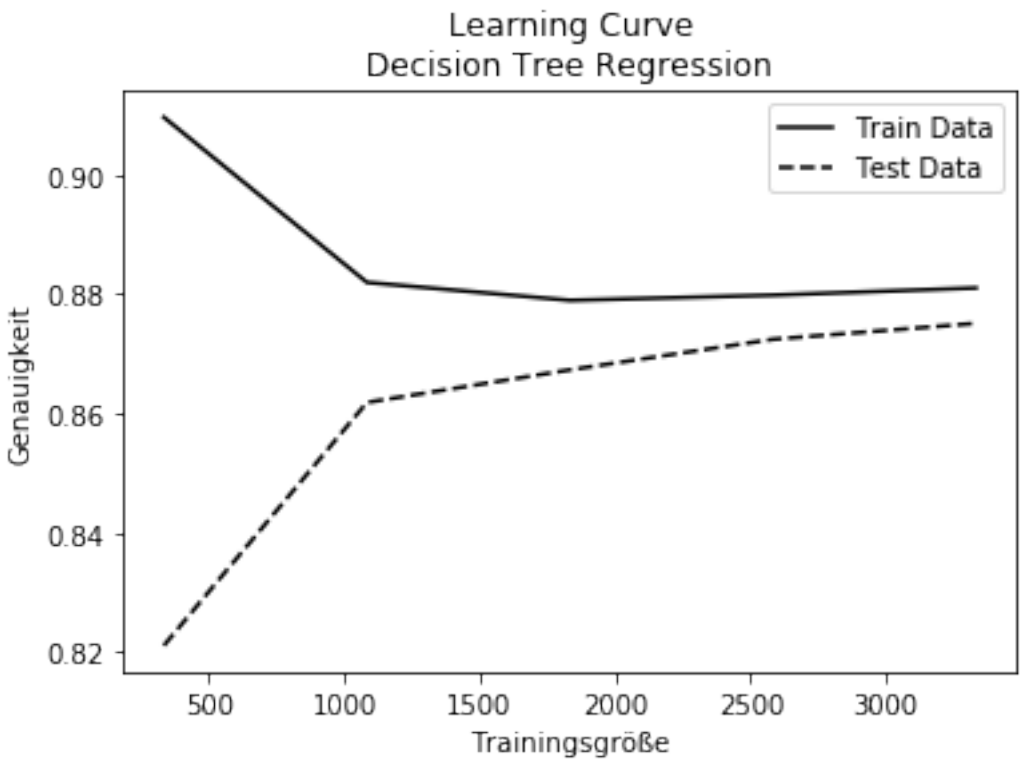
Genauigkeit der Vorhersagen auswirkt.

```
In [26]: from sklearn.model_selection import learning_curve

         # function to visualize the learning curve for a model
         # @param: model for which the learning curve should be visualized
         def draw_learning_curve(model):
             # calculate learning curve
             train_sizes_abs, train_scores, test_scores = learning_curve(model, X, y)

             # set title of the diagram
             plt.title('Learning Curve' + '\n' + get_model_name(model))
             # label y axis
             plt.ylabel('Genauigkeit')
             # label x axis
             plt.xlabel('Trainingsgröße')

             # draw curve of average training scores for each training size
             plt.plot(
                train_sizes_abs, np.mean(train_scores, axis = 1), label='Train Data',
                color='black'
             )
             # draw curve of average test scores for each training size
             plt.plot(
                train_sizes_abs, np.mean(test_scores, axis = 1), linestyle='dashed',
                label='Test Data', color='black'
             )
             # draw a legend for the both curves
             plt.legend()
             # show graph
             plt.show()

In [27]: # call function to visualize the learning curve for each model
         draw_learning_curve(lr)
         draw_learning_curve(br)
         draw_learning_curve(svr)
         draw_learning_curve(dtr)
```


Learning Curve
Lineare Regression
Train Data
Test Data
0.886
0.884
0.882
0.880
0.878
0.876
Genauigkeit
500
1000
1500
2000
2500
3000
Trainingsgröße


Learning Curve
Bayesian Regression
Train Data
Test Data
0.886
0.884
0.882
0.880
0.878
0.876
Genauigkeit
500
1000
1500
2000
2500
3000
Trainingsgröße


Learning Curve
Support Vector Regression
Train Data
Test Data
Genauigkeit
0.886
0.884
0.882
0.880
0.878
0.876
500
1000
1500
2000
2500
3000
Trainingsgröße



Im finalen Schritt kann eine Vorhersage der Zustelldauer mit jedem Modell berechnet werden. Dies wird für einen Datensatz aus dem Testset durchgeführt, welcher den Modellen unbekannt ist. Dieser wurde beim Training nicht einbezogen. Die Ergebnisse der Vorhersagen der Modelle werden tabellarisch verglichen. Zudem wird die Differenz zwischen dem berechneten und dem korrekten Wert errechnet und dargestellt.

```
In [28]: # for a clearer output in a table
         from beautifultable import BeautifulTable
         table = BeautifulTable()

         # define headline
         table.column_headers = ["", "Berechneter Wert", "Korrekter Wert", "Differenz"]

         # function to fill each row
         # @param model: model for which the entries should be written
         def add_row(model):
             table.append_row([
                 get_model_name(model), # model name
                 model.predict(X_test[:1])[0], # predict the first entry in the test set
                 y_test[:1].values[0], # correct value
                 abs(model.predict(X_test[:1])[0] - y_test[:1].values[0]) # difference between
             ])
```

```
        # fill rows
        add_row(lr)
        add_row(br)
        add_row(svr)
        add_row(dtr)

        # print results in a table
        print(table)
```

```
+------------------------------+------------------+----------------+-----------+
|                              | Berechneter Wert | korrekter Wert | Differenz |
+------------------------------+------------------+----------------+-----------+
|      Lineare Regression      |      7.376       |     8.267      |   0.891   |
+------------------------------+------------------+----------------+-----------+
|     Bayesian Regression      |      7.378       |     8.267      |   0.889   |
+------------------------------+------------------+----------------+-----------+
|  Support Vector Regression   |      7.374       |     8.267      |   0.893   |
+------------------------------+------------------+----------------+-----------+
|   Decision Tree Regression   |      7.701       |     8.267      |   0.566   |
+------------------------------+------------------+----------------+-----------+
```