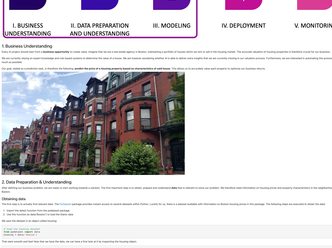
Developing AI tools - Supervised Regression

This second module of an **AI Portfolio** course describes how to develop AI solutions for generate value. This tutorial details the practice of development process with a **supervised regression** example in Python. The Data Science Life Cycle consists of the following five steps:

1. Business Understanding
2. Data Preparation & Understanding
3. Modelling
4. Deployment
5. Monitoring

In this module, we focus on the Data Wrangling approach to Machine Learning. The displayed work is a working notes of an Jupyter notebook. Code is in the Python programming language.



1. Business Understanding

Every AI project should begin with a **business opportunity** to create value. Imagine that you are a real estate agency in Boston, maintaining a portfolio of houses which on rent or sell to the housing market. The accurate valuation of housing properties is therefore crucial for our business.

We are currently relying on expert knowledge and data based systems to determine the value of a house. We are however wondering whether AI is able to deliver extra insights that we are currently missing in our valuation process. Furthermore, we are interested in automating this process as much as possible.

Our goal, related to a predictive task, is therefore the following: **predict the value of a housing property based on characteristics of a particular** The given is a publically open-land property in upstate New Hampshire.

2. Data Preparation & Understanding

Before starting any business project, we need to make certain that the data needed is available. The first important step is to collect, prepare and understand **data** that is relevant to solve our problem. We therefore need information on housing prices and property characteristics in the neighborhood of Boston.

Obtaining data

The first step is to actually find relevant data. The `Sklearn` package provides certain access to several datasets within Python. Luckily for us, there is a dataset available with information on Boston housing prices in the package. The following steps are executed to obtain this data:

1. Import the dataset function from the sklearn package
2. Load the Boston housing dataset into a variable called `boston`

We type the following in an Jupyter notebook cleaning:

```
# Import the housing dataset
from sklearn.datasets import load_boston
housing_data = load_boston()
```

Type exact amount and feed them that are from the data, we can have a first look at it by inspecting the housing objects:

```
# Inspect the housing dataset
housing_data
```

```
{'data': array([[0.00632, 18.  ,  2.31,  0.  ,  0.538,  6.575, 65.2 ,  4.09 ,
         1.  , 296.  ,  15.3 , 396.9 ,  4.98 ],
        [0.02731,  0.  ,  7.07,  0.  ,  0.469,  6.421, 78.9 ,  4.9671,
         2.  , 242.  ,  17.8 , 396.9 ,  9.14 ],
        [0.02729,  0.  ,  7.07,  0.  ,  0.469,  7.185, 61.1 ,  4.9671,
         2.  , 242.  ,  17.8 , 392.83,  4.03 ],
        [0.08829, 12.5 ,  7.87,  0.  ,  0.524,  6.012, 66.6 ,  5.5605,
         5.  , 311.  ,  15.2 , 395.6 ,  12.43 ],
        [0.19186, 12.5 ,  7.87,  0.  ,  0.524,  6.472, 54.3 ,  5.5605,
         5.  , 311.  ,  15.2 , 396.9 ,  19.31 ],
        ...
        [0.06076,  0.  ,  11.93,  0.  ,  0.573,  6.976, 91.0 ,  2.1675,
         1.  , 273.  ,  21.0 , 396.9 ,  5.64 ],
        [0.10959,  0.  ,  11.93,  0.  ,  0.573,  6.794, 89.3 ,  2.3889,
         1.  , 273.  ,  21.0 , 393.45 ,  6.48 ],
        [0.04741,  0.  ,  11.93,  0.  ,  0.573,  6.030, 80.8 ,  2.5050,
         1.  , 273.  ,  21.0 , 396.90 ,  7.88 ]],
       dtype=float64)
```

Data comes in numpy format.

The data frame has 506 cases and 14 columns on housing values in columns of Boston:

- **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 mean of 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(Bk - 0.63)^2 where Bk is the proportion of blacks by town.
- **lstat** lower status of the population (percent).
- **medv** median value of owner-occupied homes in $1000s (US).

We can inform us yourself on the variables in the **data dictionary or data descriptions problem?** Remember that the business goal is to predict the price of a housing property expressed in its characteristics. We have an indicator for the house's value, i.e. **medv** is therefore our **target** variable and all other variables are potential predictors. We have several housing characteristics that we can use to predict the target, so these are our **features**. A question that rises is a good dataset to work from with relevant information for our business problem.

Ethical considerations

Notice how the data contains the feature **chas** which measures the proportion of black people by town. Even though this feature is available, that does not mean we need to use it. Such features coding for race are a prime to need to resist because it discriminates against people of color and therefore decides to throw this feature out of our analysis.

```
# Extract the feature names
housing_feature_names = housing_data.feature_names
print(housing_feature_names)
```

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

Quality check

Before continuing it is wise also important to check the **quality** of our data. Useful features are target is one thing, but if they have a lot of missing or wrong values then they are not that valuable. Below we calculate the number of missing values for each column in our data, which is apparently equal to zero:

```
# Set the number of missing values per column
print(np.isnan(housing_data.data).sum(axis=0))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Missing not bad quality data, that's good! Let's gain some insights in the data here.

Inspect the data

We want some information on the different columns in the data by the `info()` method. This teaches us that all columns contain zero missing values (which has already known) and that all columns contain numerical values (either integer or floats).

```
# Inspect the housing dataset using the .info() method
housing_dataset.
```

```
<class 'sklearn.utils._bunch.Bunch'>
dtype: object
```

Because not high quality data, that's good! Let's gain some insights in the data here.

Inspect the data

We want some information on the different columns in the data by the `.info()` method. This teaches us that all columns contain zero missing values (which has already known) and that all columns contain numerical values (either integer or floats).

```
# Inspect the housing dataset using the .info() method
housing_dataset.
```

```
<class 'sklearn.utils._bunch.Bunch'>
dtype: object
```

We notice that many unique values exist column-wise and observe that most columns contain hundreds of unsatisfactory. Features 'chas' only contains two (as this is a dummy indicator) and 'rad' contains nine as the feature represents an index.

```
# Set the number of unique values for the category method
print(housing_df.nunique)
```

```
<bound method DataFrame.nunique of       CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  PTRATIO
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900    1  296.0     15.3
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671    2  242.0     17.8
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671    2  242.0     17.8
3  0.08829  12.5   7.87   0.0  0.524  6.012  66.6  5.5605    5  311.0     15.2
4  0.19186  12.5   7.87   0.0  0.524  6.472  54.3  5.5605    5  311.0     15.2
..     ...   ...    ...   ...    ...    ...   ...     ...  ...    ...      ...
501  0.06076   0.0  11.93   0.0  0.573  6.976  91.0  2.1675    1  273.0     21.0
502  0.10959   0.0  11.93   0.0  0.573  6.794  89.3  2.3889    1  273.0     21.0
503  0.04741   0.0  11.93   0.0  0.573  6.030  80.8  2.5050    1  273.0     21.0
504  0.07896   0.0  11.93   0.0  0.573  6.976  91.1  2.1224    1  273.0     21.0
505  0.10959   0.0  11.93   0.0  0.573  6.794  89.3  2.3889    1  273.0     21.0

     B  LSTAT  medv
0  396.9   4.98  24.0
1  396.9   9.14  21.6
2  392.83  4.03  34.7
3  395.6  12.43  33.4
4  396.9  19.31  36.2
..   ...    ...   ...
501  396.9   5.64  23.9
502  393.45  6.48  22.0
503  396.90  7.88  11.9
504  396.90  8.10  11.3
505  393.45  6.48  22.0
```

Univariate exploration

A first interesting insight is to look at the distribution of the different columns. Below we show a table with summary statistics for each column such as the mean, minimum, maximum, and quartiles. The minimum, average and maximum target values also for our example is 5.0 and 50.0 respectively. The average house price is 22.5 with a standard deviation of 9.2.

```
# Look at the summary statistics for the dataset
housing_dataset.
```

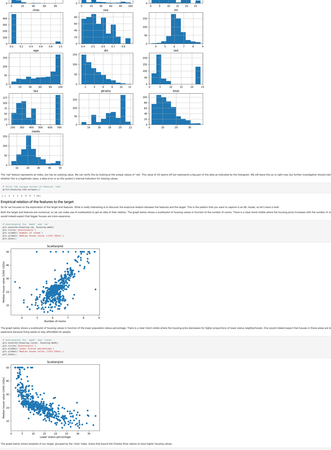
```
       CRIM          ZN       INDUS        CHAS         NOX          RM         AGE         DIST         RAD         TAX     PTRATIO           B       LSTAT        MEDV
count  506.  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean    3.  11.363636   11.136779    0.069170    0.554695    6.284634   68.574901    3.795043    9.549407  408.237154   18.455468  356.674032   12.653063   22.532806
std     8.  23.322453    6.860353    0.253994    0.115878    0.702617   28.148861    2.105710    8.707259  168.537116    2.164946   91.380860    7.141062    9.197104
min     0.   0.000000    0.460000    0.000000    0.385000    3.561000    2.900000    1.129600    1.000000  187.000000   12.600000    0.320000    1.730000    5.000000
25%     0.   0.000000    5.190000    0.000000    0.449000    5.885500   45.025000    2.100175    4.000000  279.000000   17.400000  375.377500    6.949000   17.025000
50%     0.   0.000000    9.690000    0.000000    0.538000    6.208500   77.500000    3.207450    5.000000  330.000000   19.050000  391.440000   11.360000   21.200000
75%     3.  12.083333   18.100000    0.000000    0.624000    6.623500   94.075000    5.188425    24.000000  666.000000   20.200000  396.225000   16.955750   25.000000
max    88.  100.000000   27.740000    1.000000    0.871000    8.780000  100.000000   12.126500   24.000000  711.000000   22.000000  396.900000   37.970000   50.000000
```

Table are interesting but not great visual data is always more informative. Distinguishes are a very useful tool for exploring distributions. For this we use the very popular `seaborn` module from the `matplotlib` package in Python.

```
# Import seaborn module and matplotlib package
import matplotlib.pyplot as plt, seaborn
```

Below we show histograms for each column in the Boston housing data. We observe that our target 'medv' follows a somewhat normal distribution with a slightly heavier right tail. Do you notice something weird about one of the features? Hint: look for our index.

```
# Plot histograms for all the columns
housing_dataset.hist(figsize=(20,20), bins=50)
plt.show()
```



The 'rad' feature represents an index, but that an ordinary value. We can verify this by looking at the unique values of 'rad'. This value of 24 seems off but represents a big part of the data as indicated by the histogram. We will leave this as is right now, but further investigation should clarify whether this is a legitimate value, a data error or the system's internal indicator for missing values.

```
# Print the unique values of feature 'rad'
housing_dataset['rad'].unique()
```

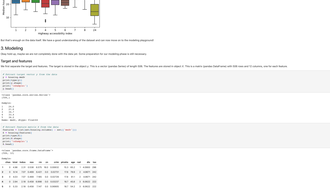
```
array([ 1.,  2.,  3.,  5.,  4.,  6.,  7.,  8.,  9., 24.])
```

Empirical relation of the features to the target

So far we focussed on the features of the target variable. Now, it is interesting to discover the empirical relation between the features and the target. This is the pattern that you want to capture in an AI. Inside, we let's have a look!

Both the target and features are numerical, so we can make use of scatterplots to get an idea of their relation. The great below shows a scatterplot of housing values in function of the number of rooms. There is a clear trend visible where the housing prices increases with the number of rooms. One interesting observation is the vertical line at the right side of the plot.

```
# Scatterplot RM versus target value
sns.scatterplot(housing_df, housing_target)
plt.ylabel('Housing Price')
plt.xlabel('Average Number of Rooms')
plt.title('Housing Price versus average number of rooms')
plt.show()
```

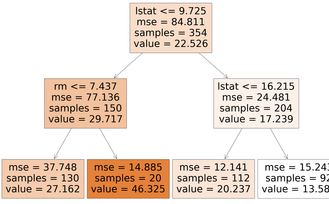


The great below shows a scatterplot of housing values in function of the lower population status percentage. There is a clear trend visible where the housing price decreases for higher proportions of lower status neighborhoods. One overall indeed expect that houses in those areas are less expensive because living there is less attractive for people.

```
# Scatterplot RM versus target value
sns.scatterplot(housing_df, housing_target)
plt.ylabel('Housing Price')
plt.xlabel('Lower Population status')
plt.title('Housing Price versus lower status')
plt.show()
```

The great below shows boxplots of our target grouped by the 'chas' index. Know that bound the Charles River seems to have higher housing values.

```
# Boxplot the 'medv' grouped by 'chas'
sns.boxplot(housing_df, housing_target)
plt.ylabel('Housing Price')
plt.xlabel('Bounding Charles River')
plt.title('Housing Price versus chas')
plt.show()
```



The great below shows boxplots of our target grouped by the 'rad' index, but in case that trend becomes apparent at first sight.

```
# Boxplot the 'medv' grouped by 'rad'
sns.boxplot(housing_df, housing_target)
plt.ylabel('Housing Price')
plt.xlabel('Index Radial Highways')
plt.title('Housing Price versus rad')
plt.show()
```


But that's enough at this data level. We have a good understanding of our data and can now move on to the modelling playground!

3. Modeling

Finally now! Up until now we are just completely done with the data part. Some preparations for our modelling phase is still necessary.

Target and features

We first separate the target and features. The target is stored in the object y. This is a vector operation. Some of weight 506. The features are stored in object X. This is a matrix. (samples (head/rows) with 506 rows and 13 columns, one for each feature.

```
# Extract target vector y from the data
y = housing_target
print(type(y))
print(y.shape)
print(y.ndim)
```

```
<class 'pandas.core.series.Series'>
(506,)
1
```

```
# Extract feature matrix X from the data
features = housing_dataset.drop(columns, axis=1 'medv')
print(type(features))
print(features.shape)
print(features.ndim)
# Display features
features
```

```
<class 'pandas.core.frame.DataFrame'>
(506, 13)
2
```

	crim	zn	indus	chas	nox	rm	age	dis	rad
0	0.00632	18.0	2.310	0.0	0.538	6.575	65.2	4.0900	1.0
1	0.02731	0.0	7.070	0.0	0.469	6.421	78.9	4.9671	2.0
2	0.02729	0.0	7.070	0.0	0.469	7.185	61.1	4.9671	2.0
3	0.08829	12.5	7.870	0.0	0.524	6.012	66.6	5.5605	5.0
4	0.19186	12.5	7.870	0.0	0.524	6.472	54.3	5.5605	5.0
...	...	...	...	...	...	...	...	...	...
501	0.06076	0.0	11.930	0.0	0.573	6.976	91.0	2.1675	1.0
502	0.10959	0.0	11.930	0.0	0.573	6.794	89.3	2.3889	1.0
503	0.04741	0.0	11.930	0.0	0.573	6.030	80.8	2.5050	1.0

Feature engineering

Feature engineering is the process of turning raw data into features that can be ours in an AI. Inside: Both our target and all features are already represented in a numerical format, interpreted by an AI. Inside: As even through feature engineering is extremely important, we don't need to do anything.

Data splitting

Before we start the modeling phase, we want that the dataset is **clean and tidy**. The Code below is used to train the AI. Inside! Write the best data is used to evaluate the model's performance. A very popular package for ML in Python is `scikit-learn`. The `train_test_split` module contains a train_test_split() function to split the data into an 80% training and 20% of the data for evaluation (see, or. We also keep the target in the dataset in the train and test set similar as stratification. Don't mind this technical term, but keeping these distributions similar is important to allow for representative training and test evaluation. Type: data is stratified on `y`, `y` and test data set `y_test`.

```
# Import the train_test_split function
from sklearn.model_selection import train_test_split
# Split the data into 80% training and 20% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                   random_state=1, stratify=y)
print("Shape of X_train: {}".format(X_train.shape))
print("Shape of y_train: {}".format(y_train.shape))
print("Shape of X_test: {}".format(X_test.shape))
print("Shape of y_test: {}".format(y_test.shape))
```

```
Shape of X_train: (354, 13)
Shape of y_train: (354,)
Shape of X_test: (152, 13)
Shape of y_test: (152,)
```

Below we show how the data looks column-wise. For observations equal to 354 of the data, while the test data contains 506 observations.

```
# Import Sklearn scaler
# Define the scaler, fit on training data, transform
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
# Transform test data using the same scaler
X_test = scaler.transform(X_test)
# Check the first 5 rows of the scaled training data
pd.DataFrame(X_train, columns=features.columns).head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad
0	-0.406	-0.48	-0.868	-0.272	-0.858	1.100	-0.351	0.557	-0.637
1	-0.419	-0.48	-0.868	-0.272	-0.858	0.787	-0.491	0.557	-0.637
2	-0.416	-0.48	-0.868	-0.272	-0.858	0.502	-0.351	0.557	-0.637
3	-0.410	-0.48	-0.868	-0.272	-0.858	0.186	-0.351	0.557	-0.637
4	-0.410	-0.48	-0.868	-0.272	-0.858	0.784	-0.351	0.557	-0.637

Simple regression tree

Now we are ready for the modeling. We want to discover the smallest structure of a tree model. For this we use the decision tree regressor from scikit-learn to model the training targets y_train based on the training features data X_train. We use a maximum tree depth equal to two to keep results intuitive and interpretable. Furthermore, we make use of the mean squared error (MSE) as loss function during training.

```
# Fit a simple regression tree
regressor = DecisionTreeRegressor(max_depth=2,
            random_state=1)
regressor.fit(X_train, y_train)
```

```
DecisionTreeRegressor
DecisionTreeRegressor(max_depth=2, random_state=1)
```

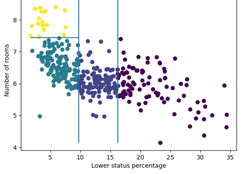
After the successful model fitting, we can explore the particular model of our regression model. A lot of information is known, but the key coefficients for each feature are the most interesting to look at. These coefficients allow to quantify and state the predicted housing price for each square area.

```
# Inspect the regression model
print(export_text(regressor,
                  feature_names=features.columns))
```

```
Old Regression Results
==============================================================================
Dep. Variable:                  y   R-squared:                       0.742
Model:                            OLS   Adj. R-squared:                  0.740
Method:                 Least Squares   F-statistic:                     359.8
Date:                Tue, 28 Jan 2020   Prob (F-statistic):          2.07e-275
Time:                        16:17:12   Log-Likelihood:                -1745.8
No. Observations:                 354   AIC:                             3541.
Df Residuals:                     340   BIC:                             3595.
Df Model:                          13
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         36.4911      5.103      7.150      0.000      26.456      46.527
crim          -0.1072      0.033     -3.207      0.001      -0.173      -0.041
zn             0.0468      0.014      3.380      0.001       0.019       0.074
indus          0.0206      0.061      0.334      0.739      -0.101       0.142
chas           2.6817      0.860      3.118      0.002       0.990       4.373
nox           -17.766      3.945     -4.505      0.000     -25.518     -10.015
rm            3.8099      0.418      9.116      0.000       2.989       4.631
age           0.0036      0.013      0.271      0.787      -0.023       0.030
dis           -1.492      0.202     -7.394      0.000      -1.889      -1.095
rad           0.2997      0.063      4.726      0.000       0.175       0.424
tax           -0.012      0.003     -4.707      0.000      -0.017      -0.007
ptratio       -0.946      0.129     -7.324      0.000      -1.201      -0.692
b             0.0093      0.002      4.819      0.000       0.005       0.013
lstat         -0.522      0.050     -10.473      0.000      -0.620      -0.424
==============================================================================
Omnibus:                      178.041   Durbin-Watson:                   1.884
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              909.977
Skew:                           1.521   Prob(JB):                    1.97e-198
Kurtosis:                       8.907   Cond. No.                         9.86e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 9.86e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Notice the coefficients for the features. Interpretation is straightforward. Notice the coefficients for the interpretation of the statistical results. The coefficients' 'const' = 3.8 indicates the predictions for a property where all features have the value of zero. The negative coefficients of the feature 'lstat' indicates that increasing the lower population status percentage decreases the predicted value (remember that lower status). The positive coefficients of 'rm' indicates that increasing the number of rooms inflates the property value (remember that more is well). More specifically exactly, increasing the number of rooms with 1 increases the value with 3.8. Interpretation of coefficients can be very useful to determine the effect of each housing feature in the predictive value.



We can also assess each feature's predictive power by calculating the so-called feature importance metric. Below we show how 'lstat' explains 71% of the predictive power and 'rm' 29%. For this tree this is not super informative, but this measure becomes really helpful when working with big complex black box models.

In [25]:
```
# Inspect feature importance values
import pandas as pd
pd.DataFrame(data = simple_tree.feature_importances_,
             index = features,
             columns = ['importance'])[simple_tree.feature_importances_ > 0]
```

Out[25]:

	importance
lstat	0.711398
rm	0.288602

Optimal regression tree

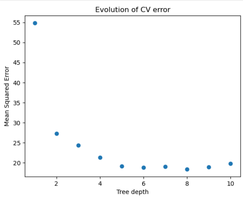
Let's now fit a slightly more complicated regression tree. We allow the tree to take a maximum depth of 10 and will search for the optimal tree depth. How do we do that? This is where the **validation** data comes in. We use part of the training data to evaluate all depth options and choose the depth that leads to the best performance. For now we define performance as the **MSE** but we come back to this later. *Sidenote: we use a more complicated setup of cross-validation (instead of one simple train-validation split) which has some advantages, but the implementation details are not a priority here.*

In [26]:
```
# Perform cross-validation to assess different tree depth options
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error

cv_grid = GridSearchCV(DecisionTreeRegressor(random_state = 1),
                       param_grid = {'max_depth':(1,2,3,4,5,6,7,8,9,10)},
                       scoring = make_scorer(mean_squared_error, greater_is_better = False),
                       cv = 5, refit = True).fit(X_train, y_train)
```

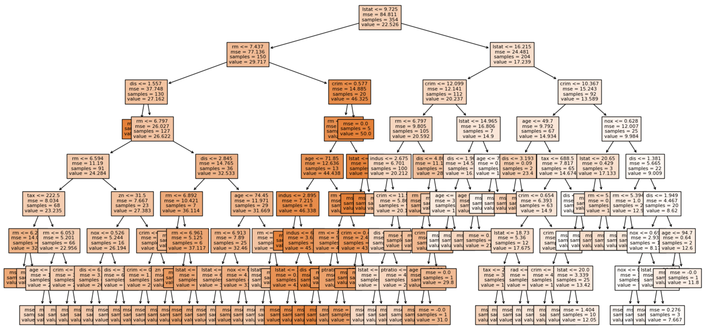
Below we visualize the MSE performance on our validation data for each tree depth. It is clear that performance improves from depth 1 to depth 6. For larger trees (depth > 6), the MSE increases again. This indicates that the tree becomes too complex and starts overfitting. We therefore pick the tree of depth 6 as optimal model.

In [27]:
```
# Visualize the accuracy performance for each tree depth
plt.scatter(x = [1,2,3,4,5,6,7,8,9,10],
            y = -cv_grid.cv_results_.get('mean_test_score'))
plt.title('Evolution of CV error')
plt.xlabel('Tree depth')
plt.ylabel('Mean Squared Error')
plt.show()
```



The optimal tree of depth 6 is visualized below. The underlying decision process can be unraveled in the exact same way as before.

In [28]:
```
# Visualize the optimal decision tree
plt.figure(figsize = (20,10))
best_tree = cv_grid.best_estimator_
tree.plot_tree(best_tree, feature_names = features, filled = True, fontsize = 8)
plt.show()
```



The feature importance scores now contain more features than in the case of our simple regression tree. We observe that 'lstat and 'rm' are still by far the most important features as together the still explain over 86% of the predictive power.

In [29]:
```
# Inspect feature importance values
pd.DataFrame(data = best_tree.feature_importances_,
             index = features,
             columns = ['importance'])).sort_values('importance', ascending = False)
```

Out[29]:

	importance
lstat	0.543211
rm	0.312331
dis	0.068612
crim	0.035725
age	0.012222
tax	0.010135
nox	0.008474
indus	0.004853
zn	0.002545
ptratio	0.001723
rad	0.000170
chas	0.000000

Model evaluation

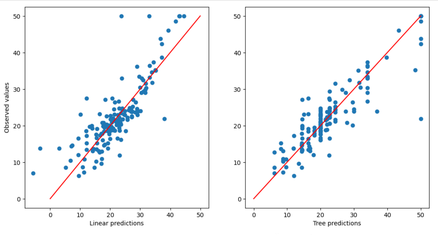
We now have two models which we can compare, namely the linear regression model and the optimal regression tree of depth 6. Let's evaluate these two models on the **test** data to assess the predictive generalization performance towards unseen data. For both models we calculate the predicted housing values of all observations in the test data.

In [30]:
```
preds_linr = lin_reg.predict(sm.add_constant(X_test))
preds_tree = best_tree.predict(X_test)
```

For evaluation purposes, we now want to compare the predictions of both models to the observed housing values in the test data. The two graphs below show scatterplots of the observed values in function of the predictions for the linear model (left) and tree (right). The blue dots should be as close as possible to the red line, because then the observed and predicted values are equal. These plots do not really say which model performs best, but teaches us two interesting facts:

- Predictions for the linear model are of a continuous nature, while the tree results in discrete predictions. A tree can by design only return as many predictions as there are leaf nodes, which is not the case for linear models.
- The linear model can result in negative value predictions in some cases. This is of course not feasible in reality and should be corrected a-posteriori.

In [31]:
```
# Compare observed and predicted values
plt.figure(figsize = (12,6))
plt.subplot(1, 2, 1)
plt.scatter(x = preds_linr,
            y = y_test)
plt.plot([0,50], [0,50], 'r')
plt.xlabel('Linear predictions')
plt.ylabel('Observed values')
plt.subplot(1, 2, 2)
plt.scatter(x = preds_tree,
            y = y_test)
plt.plot([0,50], [0,50], 'r')
plt.xlabel('Tree predictions')
plt.show()
```



We can summarize these graphs in one performance measure via for example the $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ with N the number of observations, y_i the observed targets and $\hat{y}_i$ the predicted targets. We calculate the MSE values and observe that the regression tree is performing better than the linear regression model, as lower MSE indicates better performance.

In [32]:
```
# Calculate the MSE
print('Test MSE regression tree:', mean_squared_error(y_test, preds_tree))
print('Test MSE linear regression:', mean_squared_error(y_test, preds_linr))
print('\nManual checks:')
print('RT: ', sum((y_test - preds_tree)**2) / len(y_test))
print('LR: ', sum((y_test - preds_linr)**2) / len(y_test))

Test MSE regression tree: 20.401408976801577
Test MSE linear regression: 28.743165989910162

Manual checks:
RT:  20.401408976801587
LR:  28.743165989910166
```

We can try out another measure, for example the mean absolute error $MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$. We calculate the MAE values and observe that the regression tree is again performing better than the linear regression model, as lower MAE indicates better performance.

In [33]:
```
# Calculate the MAE
from sklearn.metrics import mean_absolute_error
print('Test MAE regression tree:', mean_absolute_error(y_test, preds_tree))
print('Test MAE linear regression:', mean_absolute_error(y_test, preds_linr))
print('\nManual checks:')
print('RT: ', sum(np.abs(y_test - preds_tree)) / len(y_test))
print('LR: ', sum(np.abs(y_test - preds_linr)) / len(y_test))

Test MAE regression tree: 2.9584180284799477
Test MAE linear regression: 3.7874676576113744

Manual checks:
RT:  2.9584180284799477
LR:  3.7874676576113746
```

This analysis shows that two different metrics prefer the tree model over the linear one in the end. It is therefore clear that we should continue working with the regression tree as best AI solution for our business problem.

4. Deployment & 5. Monitoring

Once you picked a model it is time to leave the data scientist's playground and put that model in production where it can start delivering actual business value. We discuss deployment and monitoring in modules 4 and 5 of the AI4Business course, so stay tuned!

PROPHECY LABS
ctg
nannyML
AI4Business
ESF
INVESTEERT IN JOUW TOEKOMST
Europese Unie
Vlaanderen
is werk