# Predicting Continuous Target Variables with Regression Analysis

Throughout the previous chapters, you learned a lot about the main concepts behind **supervised learning** and trained many different models for classification tasks to predict group memberships or categorical variables. In this chapter, we will dive into another subcategory of supervised learning: **regression analysis**.

Regression models are used to predict target variables on a continuous scale, which makes them attractive for addressing many questions in science as well as applications in industry, such as understanding relationships between variables, evaluating trends, or making forecasts. One example would be predicting the sales of a company in future months.

In this chapter, we will discuss the main concepts of regression models and cover the following topics:

- Exploring and visualizing datasets
- Looking at different approaches to implement linear regression models
- Training regression models that are robust to outliers
- Evaluating regression models and diagnosing common problems
- Fitting regression models to nonlinear data

# Introducing linear regression

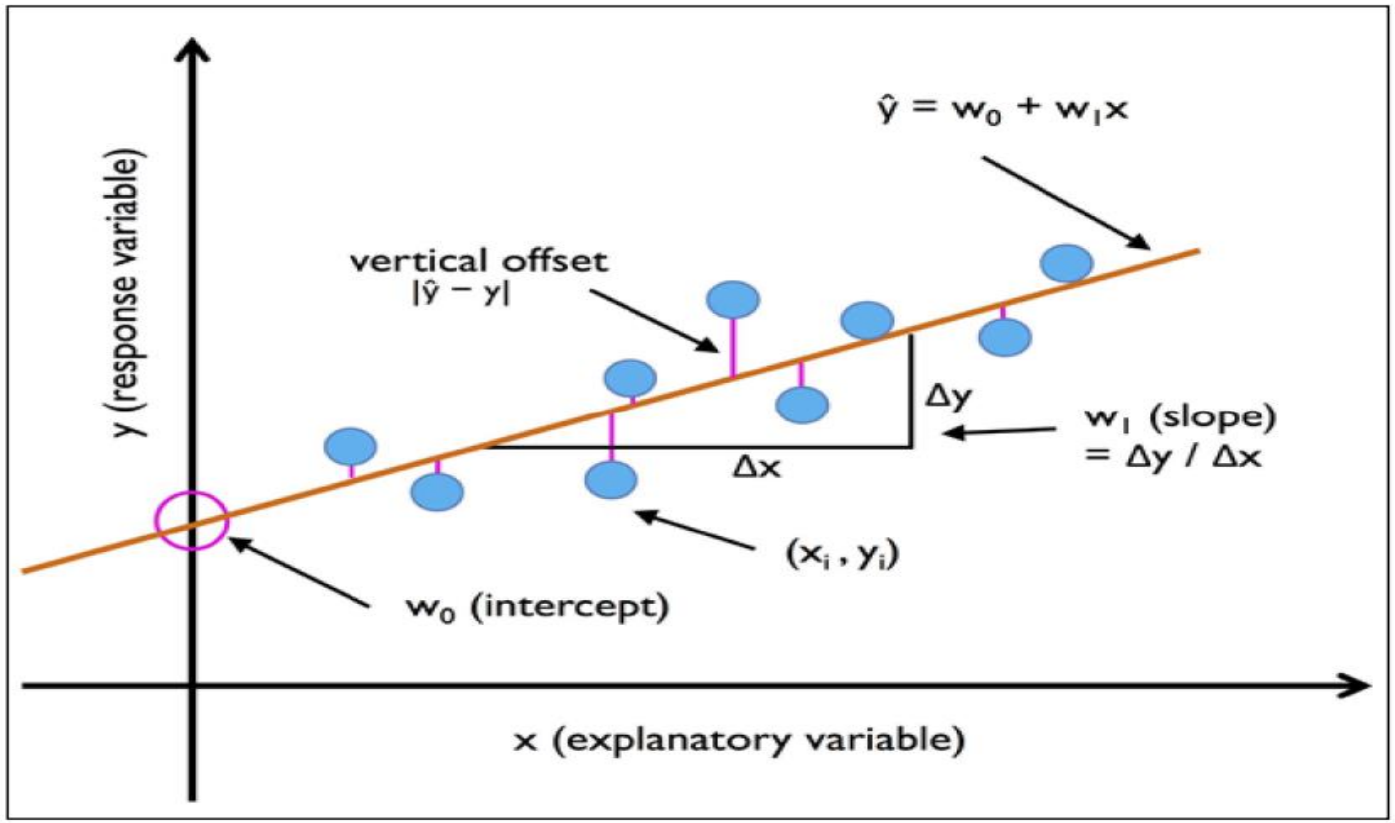
The goal of linear regression is to model the relationship between one or multiple features and a continuous target variable. As discussed in *Chapter 1, Giving Computers the Ability to Learn from Data*, regression analysis is a subcategory of supervised machine learning. In contrast to classification—another subcategory of supervised learning—regression analysis aims to predict outputs on a continuous scale rather than categorical class labels.

# Simple linear regression

The goal of simple (**univariate**) linear regression is to model the relationship between a single feature (**explanatory variable** $x$) and a continuous valued **response** (**target variable** $y$). The equation of a linear model with one explanatory variable is defined as follows:

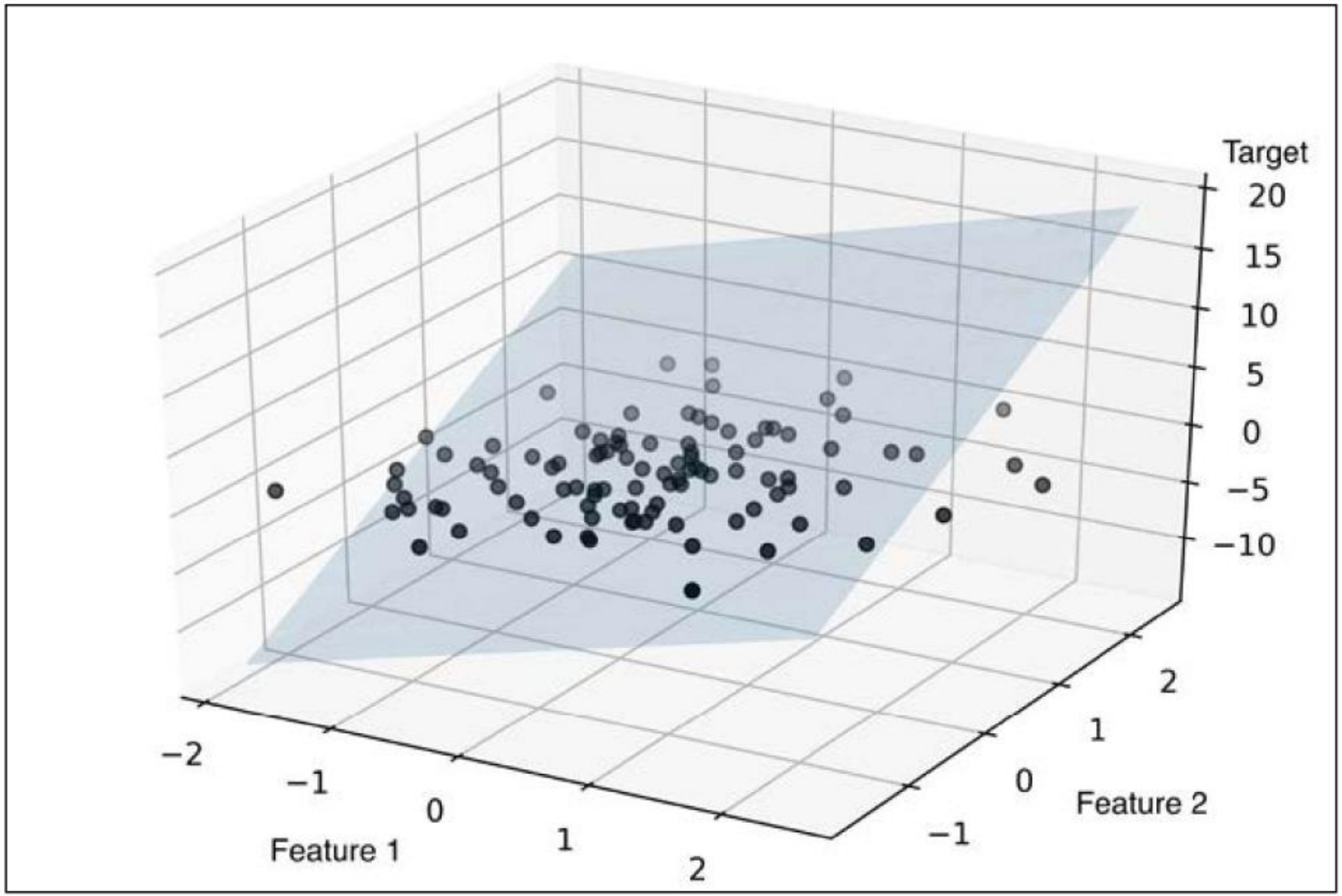
$$y = w_0 + w_1 x$$

Here, the weight $w_0$ represents the $y$-axis intercept and $w_1$ is the weight coefficient of the explanatory variable. Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset.




This best-fitting line is also called the **regression line**, and the vertical lines from the regression line to the sample points are the so-called **offsets** or **residuals**—the errors of our prediction.

The following figure shows how the two-dimensional, fitted hyperplane of a multiple linear regression model with two features could look:



# Exploring the Housing dataset

## Loading the Housing dataset into a data frame

The features of the 506 samples in the Housing dataset are summarized here, taken from the original source that was previously shared on `https://archive.ics.uci.edu/ml/datasets/Housing`:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://raw.githubusercontent.com/rasbt/'
...                  'python-machine-learning-book-2nd-edition'
...                  '/master/code/ch10/housing.data.txt',
...                  header=None,
...                  sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

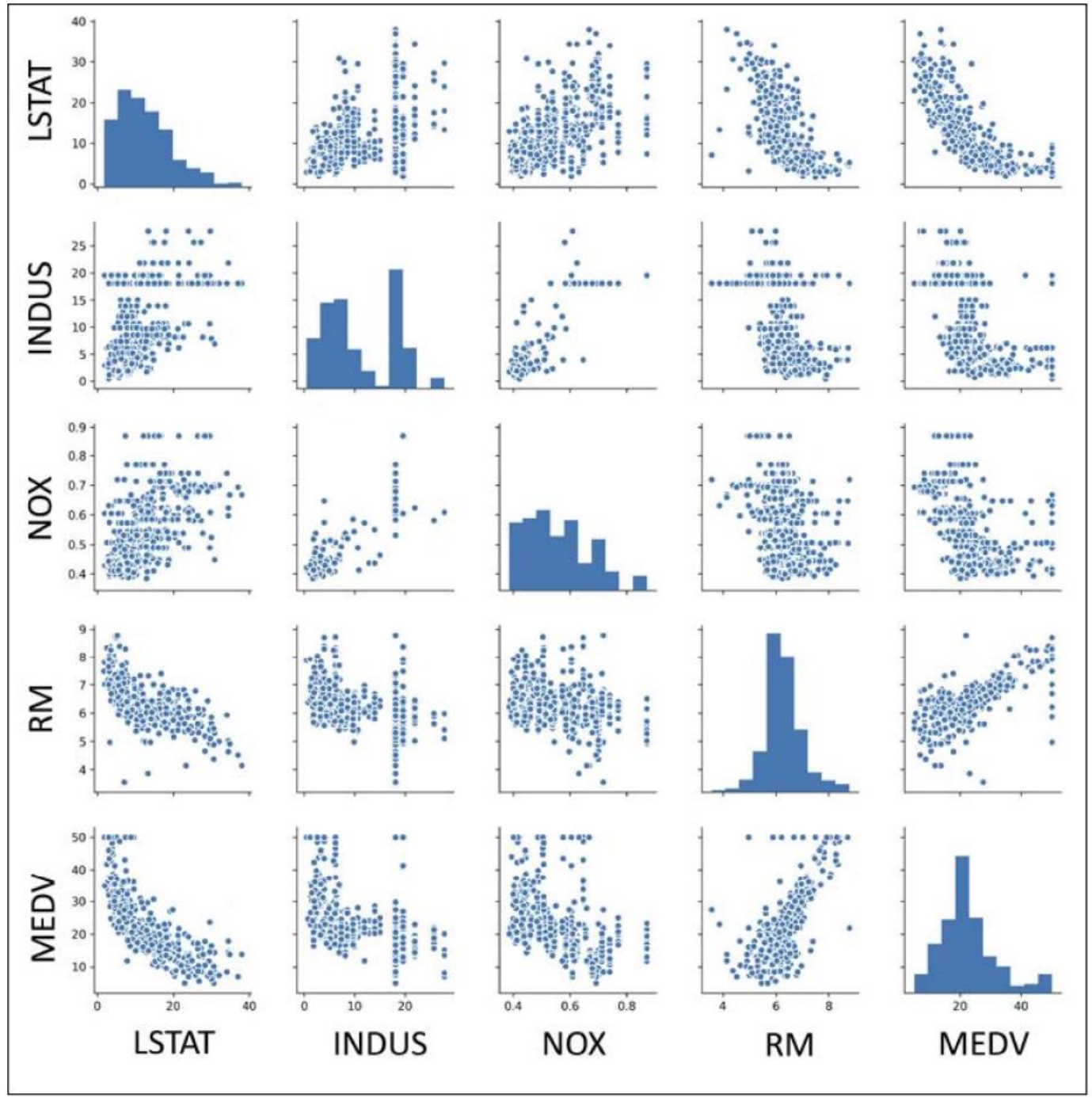
| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | MEDV |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1 | 296.0 | 15.3 | 396.90 | 4.98 | 24.0 |
| **1** | 0.02731 | 0.0 | 7.07 | 0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2 | 242.0 | 17.8 | 396.90 | 9.14 | 21.6 |
| **2** | 0.02729 | 0.0 | 7.07 | 0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2 | 242.0 | 17.8 | 392.83 | 4.03 | 34.7 |
| **3** | 0.03237 | 0.0 | 2.18 | 0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3 | 222.0 | 18.7 | 394.63 | 2.94 | 33.4 |
| **4** | 0.06905 | 0.0 | 2.18 | 0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3 | 222.0 | 18.7 | 396.90 | 5.33 | 36.2 |

# Visualizing the important characteristics of a dataset

**Exploratory Data Analysis** (**EDA**) is an important and recommended first step prior to the training of a machine learning model. In the rest of this section, we will use some simple yet useful techniques from the graphical EDA toolbox that may help us to visually detect the presence of outliers, the distribution of the data, and the relationships between features.

First, we will create a **scatterplot matrix** that allows us to visualize the pair-wise correlations between the different features in this dataset in one place. To plot the scatterplot matrix, we will use the `pairplot` function from the Seaborn library (`http://stanford.edu/~mwaskom/software/seaborn/`), which is a Python library for drawing statistical plots based on Matplotlib.

```
>>> impor
>>> impor
>>> cols
>>> sns.p
>>> plt.t
>>> plt.s
```



# Looking at relationships using a correlation matrix

In the previous section, we visualized the data distributions of the Housing dataset variables in the form of histograms and scatter plots. Next, we will create a correlation matrix to quantify and summarize linear relationships between variables. A correlation matrix is closely related to the covariance matrix that we have seen in the section about **Principal Component Analysis** (**PCA**) in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. Intuitively, we can interpret the correlation matrix as a rescaled version of the covariance matrix. In fact, the correlation matrix is identical to a covariance matrix computed from standardized features.

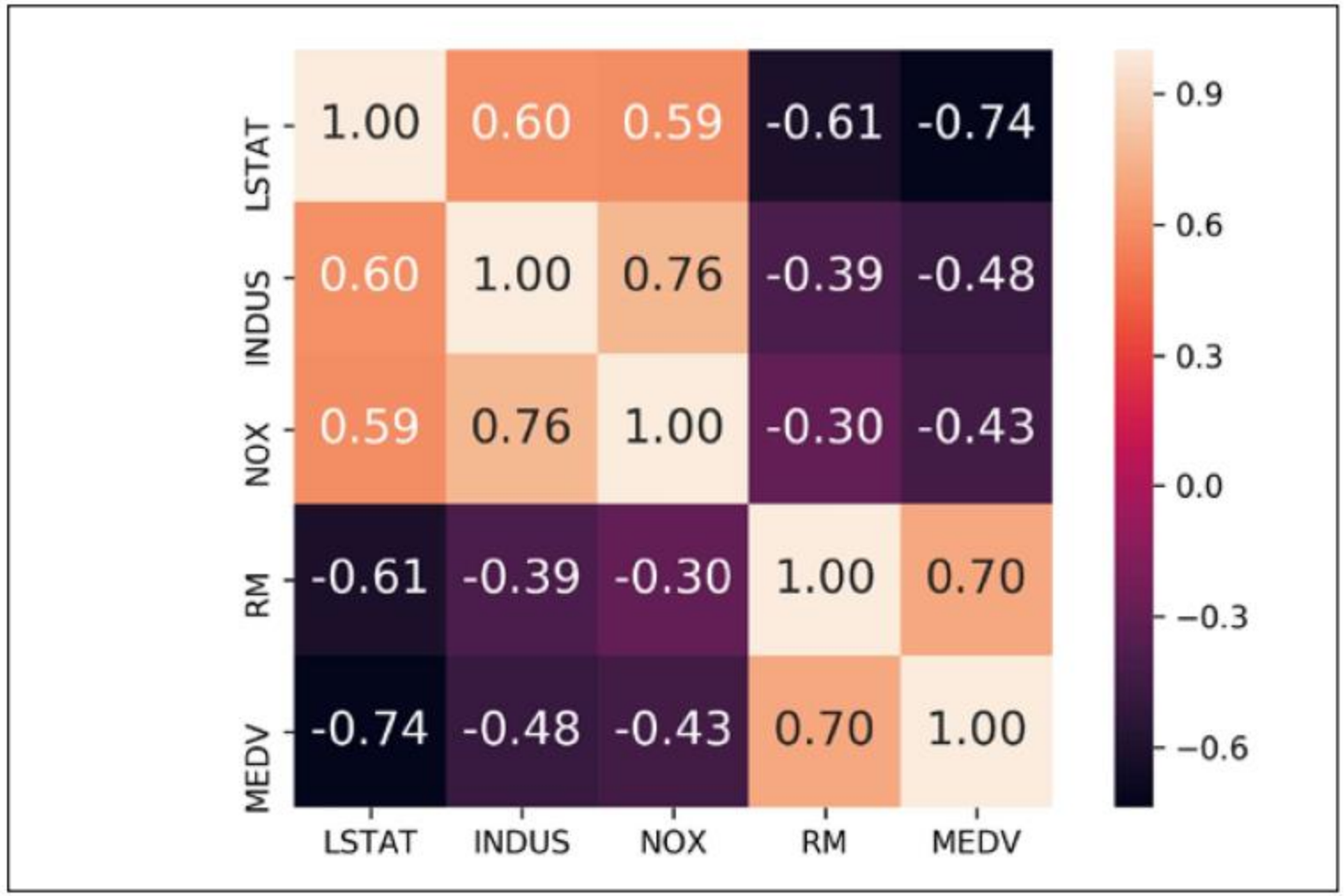
The correlation matrix is a square matrix that contains the **Pearson product-moment correlation coefficient** (often abbreviated as **Pearson's r**), which measure the linear dependence between pairs of features. The correlation coefficients are in the range -1 to 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$. As mentioned previously, Pearson's correlation coefficient can simply be calculated as the covariance between two features $x$ and $y$ (numerator) divided by the product of their standard deviations (denominator):

$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)} - \mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Here, $\mu$ denotes the sample mean of the corresponding feature, $\sigma_{xy}$ is the covariance between the features *x* and *y*, and $\sigma_x$ and $\sigma_y$ are the features' standard deviations.

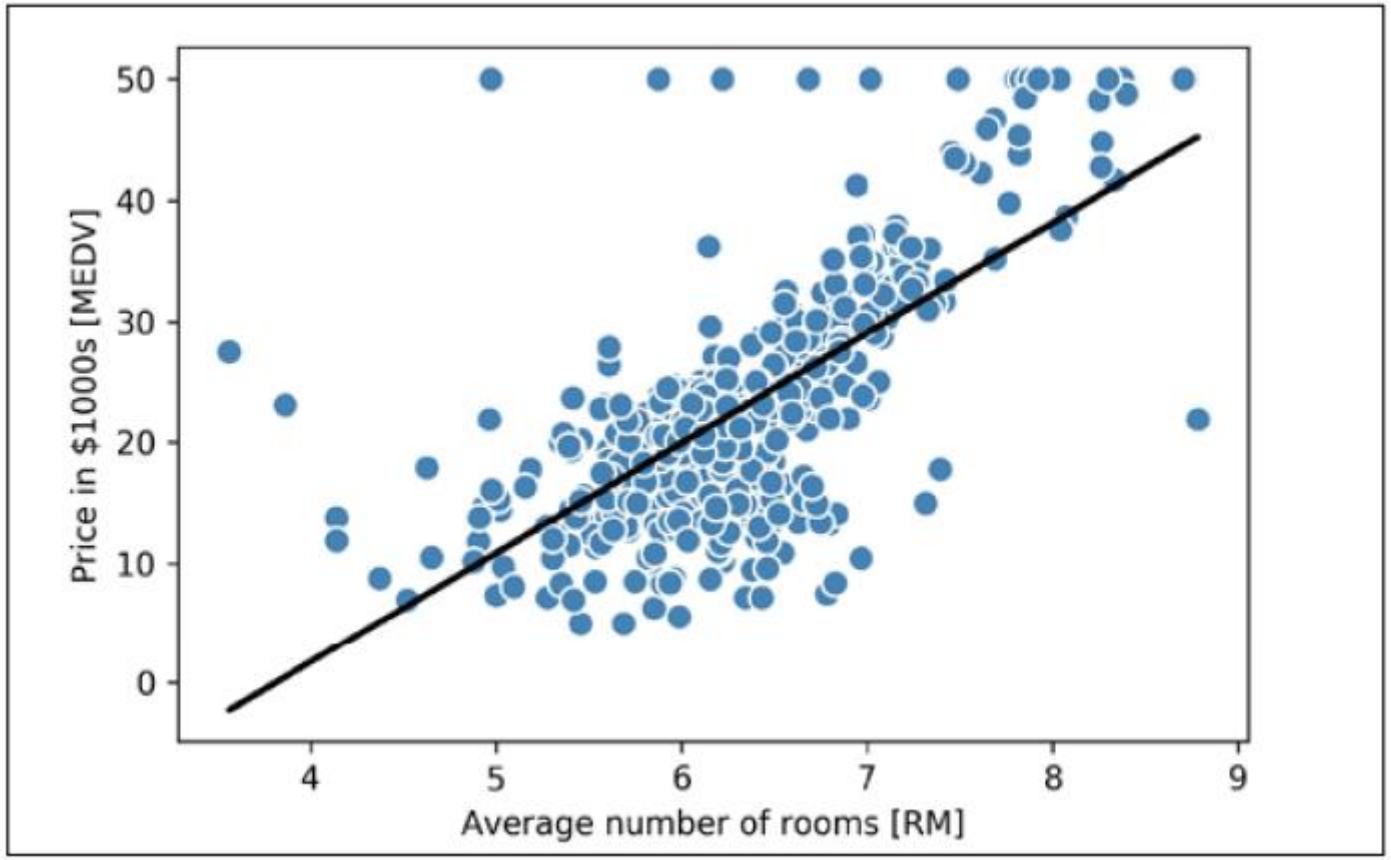
In the following code examp
feature columns that we pre
use Seaborn's `heatmap` funct

```
>>> import numpy as n
>>> cm = np.corrcoef(
>>> sns.set(font_scal
>>> hm = sns.heatmap(
...                  cbar=T
...                  annot=
...                  square
...                  fmt='.
...                  annot_
...                  ytickl
...                  xtickl
>>> plt.show()
```



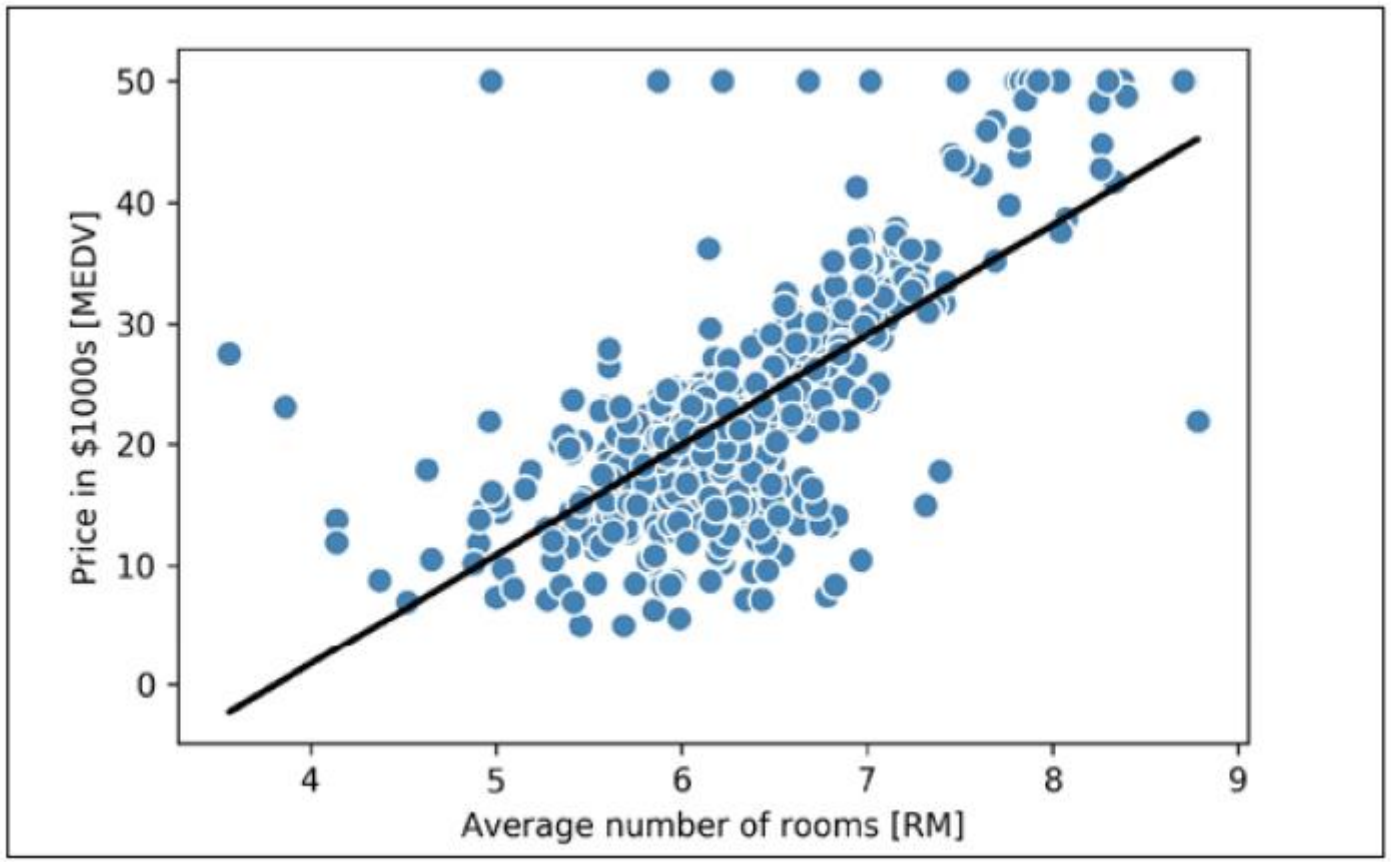
# Estimating coefficient of a regression model via scikit-learn

```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
 >>> from sklearn.linear_mod
 >>> slr = LinearRegression
 >>> slr.fit(X, y)
 >>> print('Slope: %.3f' % s
 Slope: 9.102
 >>> print('Intercept: %.3f
 Intercept: -34.671
```



```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y,
...     plt.plot(X, model
...     return None

>>> lin_regplot(X, y, sl
>>> plt.xlabel('Average
>>> plt.ylabel('Price in
>>> plt.show()
```



# Evaluating the performance of linear regression models

```
>>> from sklearn.model_selection import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...         X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Another useful quantitative measure of a model's performance is the so-called **Mean Squared Error** (**MSE**), which is simply the averaged value of the SSE cost that we minimized to fit the linear regression model. The MSE is useful to compare different regression models or for tuning their parameters via grid search and cross-validation, as it normalizes the SSE by the sample size:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Let's compute the MSE of our training and test predictions:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
...         mean_squared_error(y_train, y_train_pred),
...         mean_squared_error(y_test, y_test_pred)))
MSE train: 19.958, test: 27.196
```

We see that the MSE on the training set is 19.96, and the MSE of the test set is much larger, with a value of 27.20, which is an indicator that our model is overfitting the training data.

Sometimes it may be more useful to report the **coefficient of determination** ($R^2$), which can be understood as a standardized version of the MSE, for better interpretability of the model's performance. Or in other words, $R^2$ is the fraction of response variance that is captured by the model. The $R^2$ value is defined as:

$$R^2 = 1 - \frac{SSE}{SST}$$

Here, SSE is the sum of squared errors and SST is the total sum of squares:

$$SST = \sum_{i=1}^{n} \left( y^{(i)} - \mu_y \right)^2$$

In other words, SST is simply the variance of the response.

Let us quickly show that $R^2$ is indeed just a rescaled version of the MSE:

$$R^2 = 1 - \frac{SSE}{SST}$$

$$1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}$$

$$1 - \frac{MSE}{Var(y)}$$

For the training dataset, the $R^2$ is bounded between 0 and 1, but it can become negative for the test set. If $R^2 = 1$, the model fits the data perfectly with a corresponding $MSE = 0$.

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 train: %.3f, test: %.3f' %
...       (r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
R^2 train: 0.765, test: 0.673
```

# Fitting a robust regression model using RANSAC

Linear regression models can be heavily impacted by the presence of outliers. In certain situations, a very small subset of our data can have a big effect on the estimated model coefficients. There are many statistical tests that can be used to detect outliers

As an alternative to throwing out outliers, we will look at a robust method of regression using the **RANdom SAmple Consensus** (**RANSAC**) algorithm, which fits a regression model to a subset of the data, the so-called **inliers**.

We can summarize the iterative RANSAC algorithm as follows:

1. Select a random number of samples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
3. Refit the model using all inliers.
4. Estimate the error of the fitted model versus the inliers.
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations were reached; go back to step 1 otherwise.

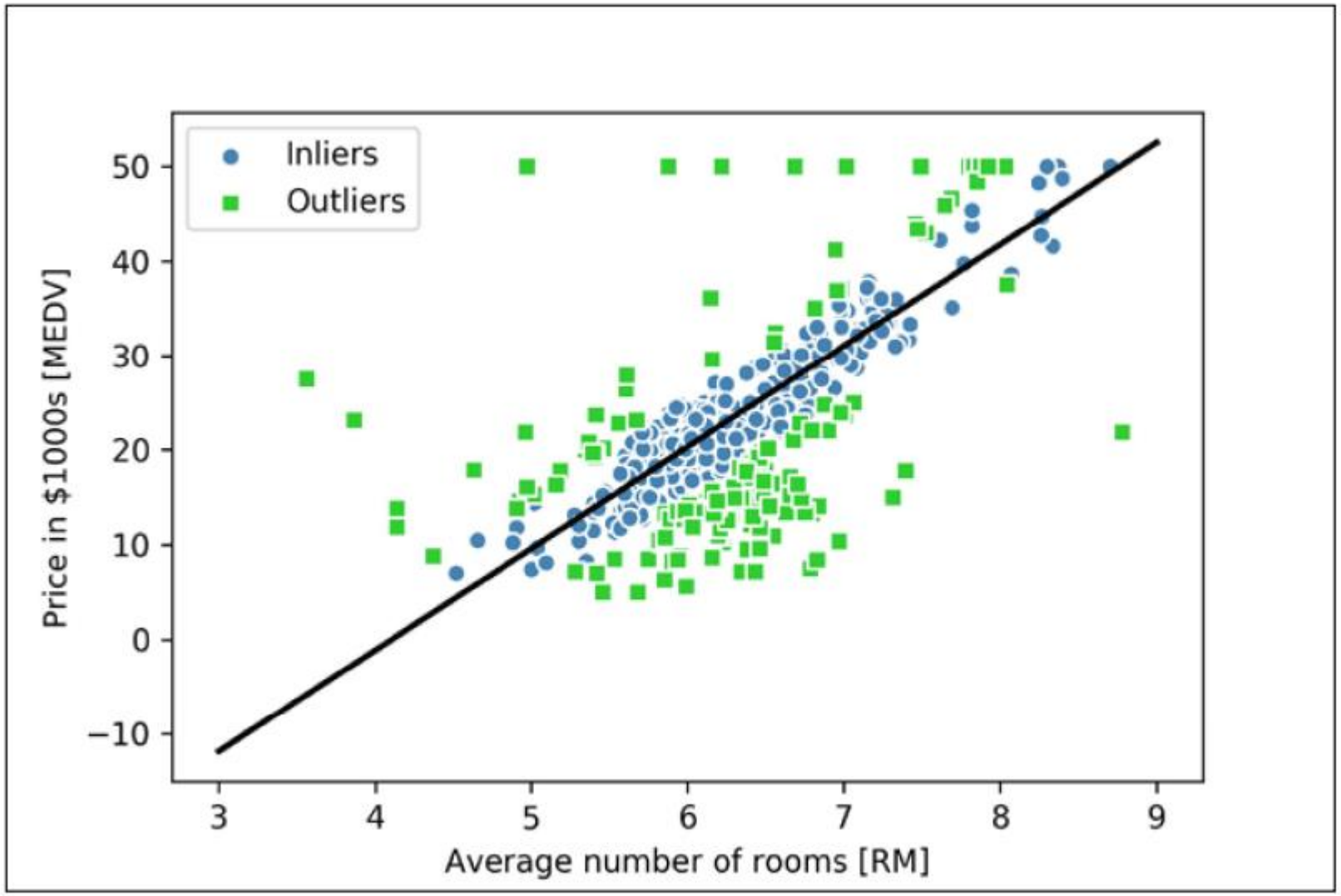
Let us now wrap our linear model in the RANSAC algorithm using scikit-learn's `RANSACRegressor` class:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...                          max_trials=100,
...                          min_samples=50,
...                          loss='absolute_loss',
...                          residual_threshold=5.0,
...                          random_state=0)
>>> ransac.fit(X, y)
```

By default, scikit-learn uses the **MAD** estimate to select the inlier threshold, where MAD stands for the **Median Absolute Deviation** of the target values `y`. However, the choice of an appropriate value for the inlier threshold is problem-specific, which is one disadvantage of RANSAC. Many different approaches have been developed in recent years to select a good inlier threshold automatically

After we fit the RANSAC model, let's obtain the inliers and outliers from the fitted RANSAC-linear regression model and plot them together with the linear fit:

```
>>> inlier_mask =
>>> outlier_mask
>>> line_X = np.a
>>> line_y_ransac
>>> plt.scatter(X
...                c
...                m
>>> plt.scatter(X
...                c
...                m
>>> plt.plot(line_
>>> plt.xlabel('A
>>> plt.ylabel('P
>>> plt.legend(lo
>>> plt.show()
```



When we print the slope and intercept of the model by executing the following code, we can see that the linear regression line is slightly different from the fit that we obtained in the previous section without using RANSAC:

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 10.735
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -44.089
```

# Using regularized methods for regression

regularization is one approach to tackle the problem of overfitting by adding additional information, and thereby shrinking the parameter values of the model to induce a penalty against complexity. The most popular approaches to regularized linear regression are the so-called **Ridge Regression**, **Least Absolute Shrinkage and Selection Operator** (**LASSO**), and **Elastic Net**.

Ridge regression is an L2 penalized model where we simply add the squared sum of the weights to our least-squares cost function:

$$J(w)_{Ridge} = \sum_{i=1}^{n} \left( y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \| w \|_2^2$$

Here:

$$L2: \quad \lambda \| w \|_2^2 = \lambda \sum_{j=1}^{m} w_j{}^2$$

By increasing the value of hyperparameter $\lambda$, we increase the regularization strength and shrink the weights of our model. Please note that we don't regularize the intercept term $w_0$.

An alternative approach that can lead to sparse models is LASSO. Depending on the regularization strength, certain weights can become zero, which also makes LASSO useful as a supervised feature selection technique:

$$J(w)_{LASSO} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \| w \|_1$$

Here:

$$L1 : \lambda \| w \|_1 = \lambda \sum_{j=1}^{m} \left| w_j \right|$$

However, a limitation of LASSO is that it selects at most $n$ variables if $m>n$. A compromise between Ridge regression and LASSO is Elastic Net, which has an L1 penalty to generate sparsity and an L2 penalty to overcome some of the limitations of LASSO, such as the number of selected variables:

$$J(w)_{ElasticNet} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} w_j{}^2 + \lambda_2 \sum_{j=1}^{m} \left|w_j\right|$$

A Ridge regression model can be initialized via:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Note that the regularization strength is regulated by the parameter `alpha`, which is similar to the parameter $\lambda$. Likewise, we can initialize a LASSO regressor from the `linear_model` submodule:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Lastly, the `ElasticNet` implementation allows us to vary the L1 to L2 ratio:

```
>>> from sklearn.linear_model import ElasticNet
>>> elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

For example, if we set the `l1_ratio` to 1.0, the `ElasticNet` regressor would be equal to LASSO regression.

# Turning a linear regression model into a curve – polynomial regression

In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$$

Here, $d$ denotes the degree of the polynomial. Although we can use polynomial regression to model a nonlinear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients $w$.

# Adding polynomial terms using scikit-learn

1. Add a second degree polynomial term:

```
from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([ 258.0, 270.0, 294.0, 320.0, 342.0,
...               368.0, 396.0, 446.0, 480.0, 586.0])\
...               [:, np.newaxis]
>>> y = np.array([ 236.4, 234.4, 252.8, 298.6, 314.2,
...                342.2, 360.8, 368.0, 391.2, 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

2. Fit a simple linear

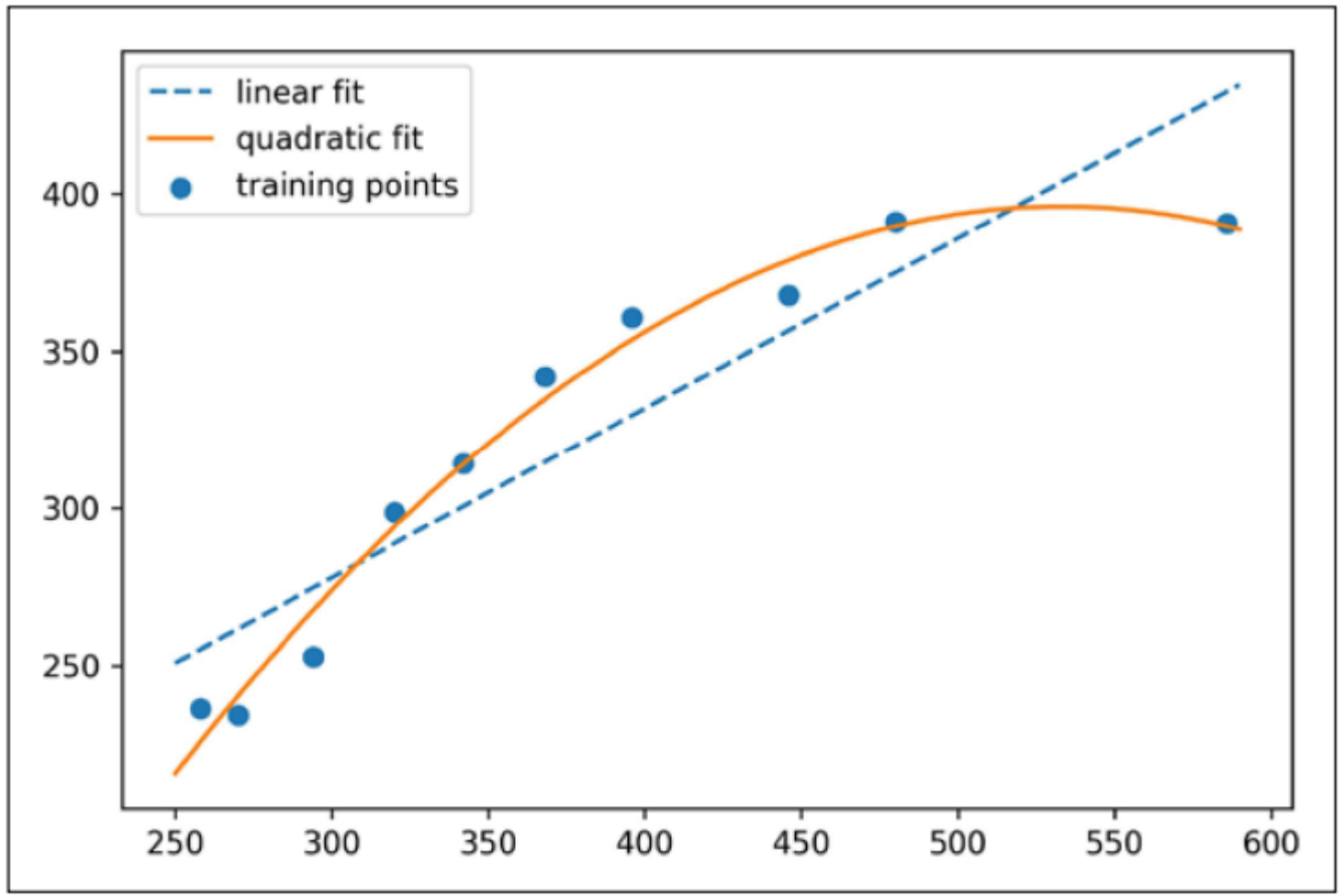
```
>>> lr.fit(X, y
>>> X_fit = np.
>>> y_lin_fit =
```

3. Fit a multiple regre
regression:

```
>>> pr.fit(X_qua
>>> y_quad_fit =
```

4. Plot the results:

```
>>> plt.scatter(
>>> plt.plot(X_f
...               lab
>>> plt.plot(X_f
...               lab
>>> plt.legend(l
>>> plt.show()
```



```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Training MSE linear: %.3f, quadratic: %.3f' % (
...         mean_squared_error(y, y_lin_pred),
...         mean_squared_error(y, y_quad_pred)))
Training MSE linear: 569.780, quadratic: 61.330
>>> print('Training  R^2 linear: %.3f, quadratic: %.3f' % (
...         r2_score(y, y_lin_pred),
...         r2_score(y, y_quad_pred)))
Training  R^2 linear: 0.832, quadratic: 0.982
```

# Modeling nonlinear relationships in the Housing dataset

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values

>>> regr = LinearRegression()

# create quadratic features
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

# fit features
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]

>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))

>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))

>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))
```

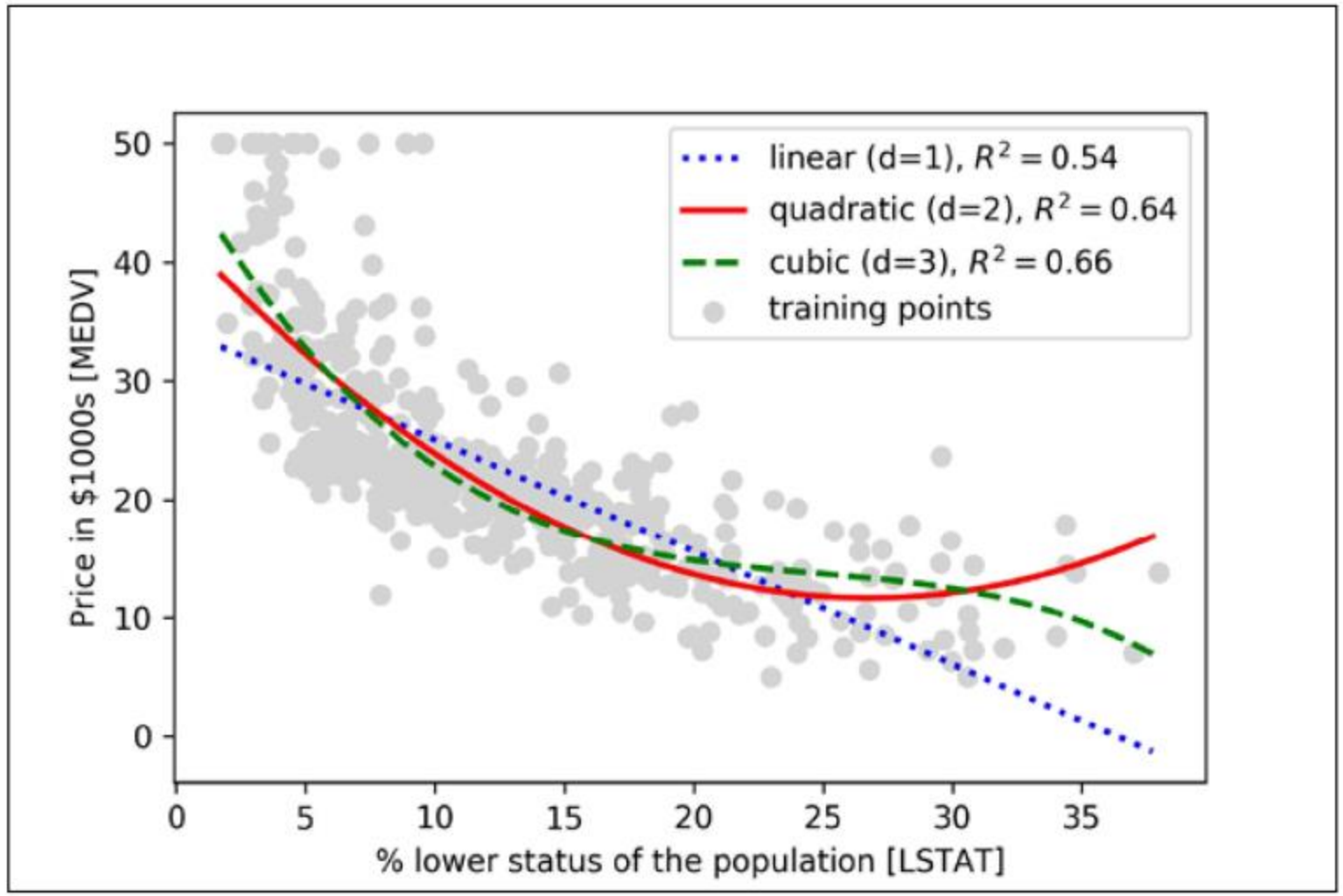
```
>>> plt.scatter(X, y, label='training points', color='lightgray')

>>> plt.plot(X_fit, y_lin_fit,
...           label='linear (d=1), $R^2=%.2f$' % linear_r2,
...           color='blue',
...           lw=2,
...           linestyle=':')

>>> plt.plot(X_fit, y_quad_fit,
...           label='quadratic (d=2), $R^2=%.2f$' % quadratic_r2,
...           color='red',
...           lw=2,
...           linestyle='-')

>>> plt.plot(X_fit, y_cubic_fit,
...           label='cubic (d=3), $R^2=%.2f$' % cubic_r2,
...           color='green',
...           lw=2,
...           linestyle='--')

>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000s [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```


linear (d=1), $R^2 = 0.54$
quadratic (d=2), $R^2 = 0.64$
cubic (d=3), $R^2 = 0.66$
training points
Price in $1000s [MEDV]
% lower status of the population [LSTAT]
0
10
20
30
40
50
0
5
10
15
20
25
30
35

# Dealing with nonlinear relationships using random forests

In this section, we are going to take a look at **random forest** regression, which is conceptually different from the previous regression models in this chapter. A random forest, which is an ensemble of multiple **decision trees**, can be understood as the sum of piecewise linear functions in contrast to the global linear and polynomial regression models that we discussed previously. In other words, via the decision tree algorithm, we are subdividing the input space into smaller regions that become more *manageable*.

# Decision tree regression

An advantage of the decision tree algorithm is that it does not require any transformation of the features if we are dealing with nonlinear data. We remember from *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, that we grow decision tree by iteratively splitting its nodes until the leaves are pure or a stopping criterion is satisfied. When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the **Information Gain** (**IG**), which can be defined as follows for a binary split:

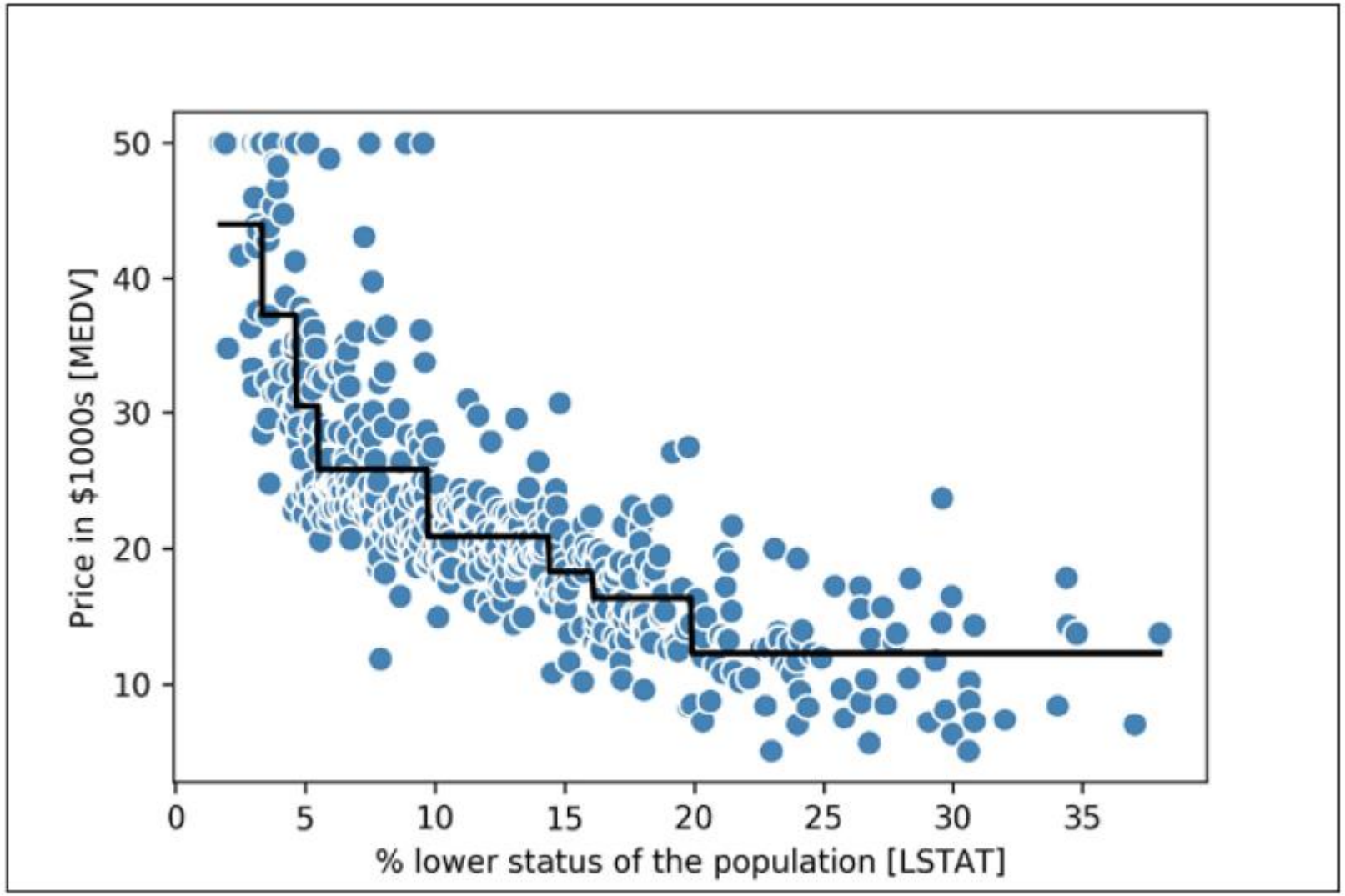
$$IG\left(D_p, x_i\right) = I\left(D_p\right) - \frac{N_{left}}{N_p} I\left(D_{left}\right) - \frac{N_{right}}{N_p} I\left(D_{right}\right)$$

$$I\left(t\right) = MSE\left(t\right) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t\right)^2$$

Here, $N_t$ is the number of training samples at node $t$, $D_t$ is the training subset at node $t$, $y^{(i)}$ is the true target value, and $\hat{y}_t$ is the predicted target value (sample mean):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

```
>>> from sklearn.tre
>>> X = df[['LSTAT']
>>> y = df['MEDV'].v
>>> tree = DecisionT
>>> tree.fit(X, y)
>>> sort_idx = X.fla
>>> lin_regplot(X[so
>>> plt.xlabel('% lo
>>> plt.ylabel('Pric
>>> plt.show()
```



As we can see in the resulting plot, the decision tree captures the general trend in the data. However, a limitation of this model is that it does not capture the continuity and differentiability of the desired prediction. In addition, we need to be careful about choosing an appropriate value for the depth of the tree to not overfit or underfit the data; here, a depth of three seemed to be a good choice:

# Random forest regression

random forest algorithm is an ensemble technique that combines multiple decision trees. A random forest usually has a better generalization performance than an individual decision tree due to randomness, which helps to decrease the model's variance. Other advantages of random forests are that they are less sensitive to outliers in the dataset and don't require much parameter tuning. The only parameter in random forests that we typically need to experiment with is the number of trees in the ensemble. The basic random forest algorithm for regression is almost identical to the random forest algorithm for classification that we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, the only difference is that we use the MSE criterion to grow the individual decision trees, and the predicted target variable is calculated as the average prediction over all decision trees.

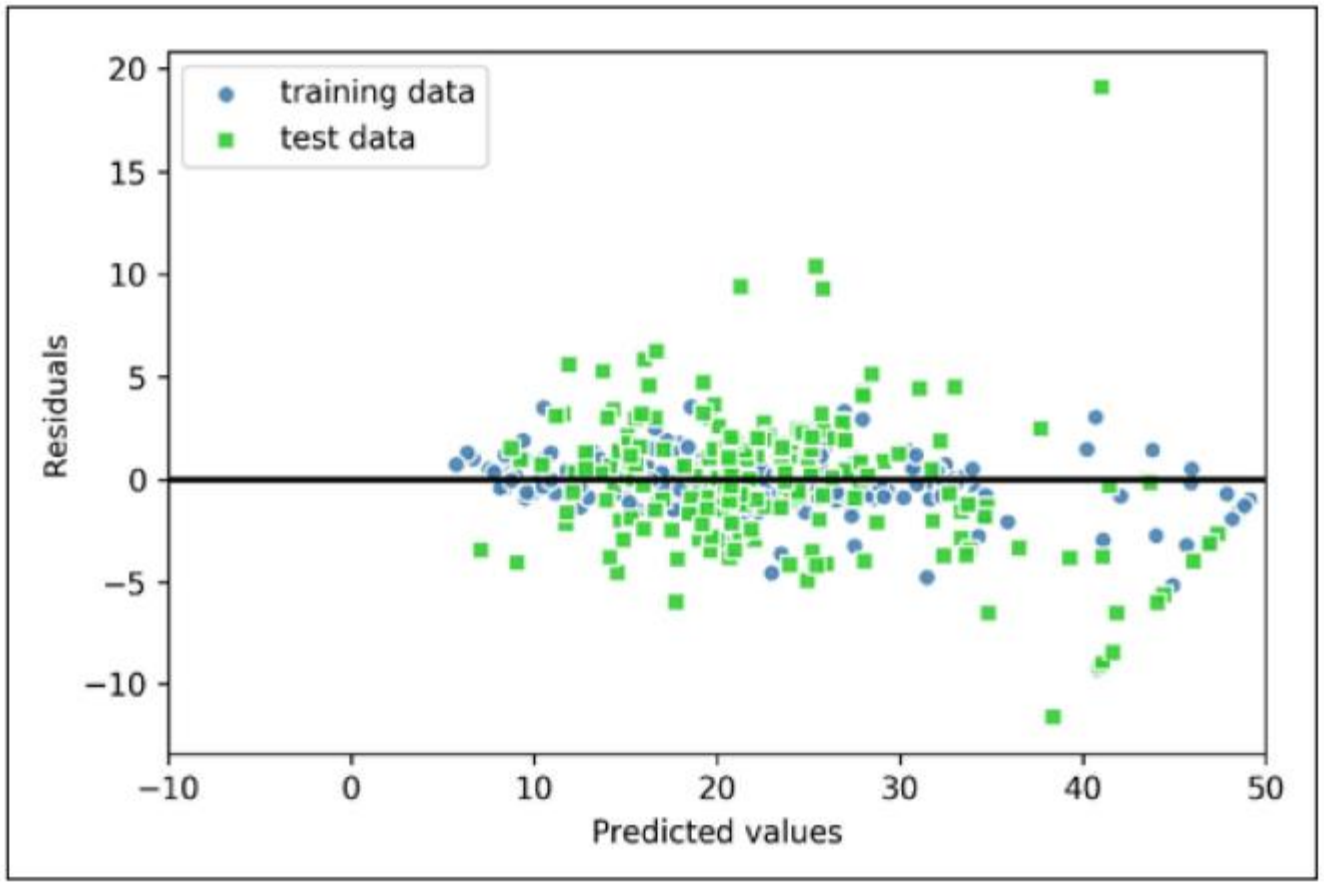
Now, let's use all features in the Housing dataset to fit a random forest regression model on 60 percent of the samples and evaluate its performance on the remaining 40 percent. The code is as follows:

```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test =\
...       train_test_split(X, y,
...                        test_size=0.4,
...                        random_state=1)

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(n_estimators=1000,
...                                criterion='mse',
...                                random_state=1,
...                                n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> print('MSE train: %.3f, test: %.3f' % (
...        mean_squared_error(y_train, y_train_pred),
...        mean_squared_error(y_test, y_test_pred)))
MSE train: 1.642, test: 11.052
>>> print('R^2 train: %.3f, test: %.3f' % (
...        r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
R^2 train: 0.979, test: 0.878
```

Lastly, let us also take a look at the residuals of the prediction:

```
>>> plt.scatter(y_train_pred
...                 y_tr
...                 c='s
...                 edge
...                 mark
...                 s=35
...                 alph
...                 labe
>>> plt.scatter(y_te
...                 y_te
...                 c='l
...                 edge
...                 mark
...                 s=35
...                 alph
...                 labe
>>> plt.xlabel('Pred
>>> plt.ylabel('Resi
>>> plt.legend(loc='
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
>>> plt.xlim([-10, 50])
>>> plt.show()
```



# Thank You !!!