9

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. They also have applications in industry, such as understanding relationships between variables, evaluating trends, or making forecasts. One example is 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 implementing 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. In contrast to classification—a different subcategory of supervised learning—regression analysis aims to predict outputs on a continuous scale rather than categorical class labels.

In the following subsections, you will be introduced to the most basic type of linear regression, **simple linear regression**, and understand how to relate it to the more general, multivariate case (linear regression with multiple features).

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 **target** (**response variable**, y). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_1 x + b$$

Here, the parameter (bias unit), b, 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.

Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the training examples, as shown in *Figure 9.1*:

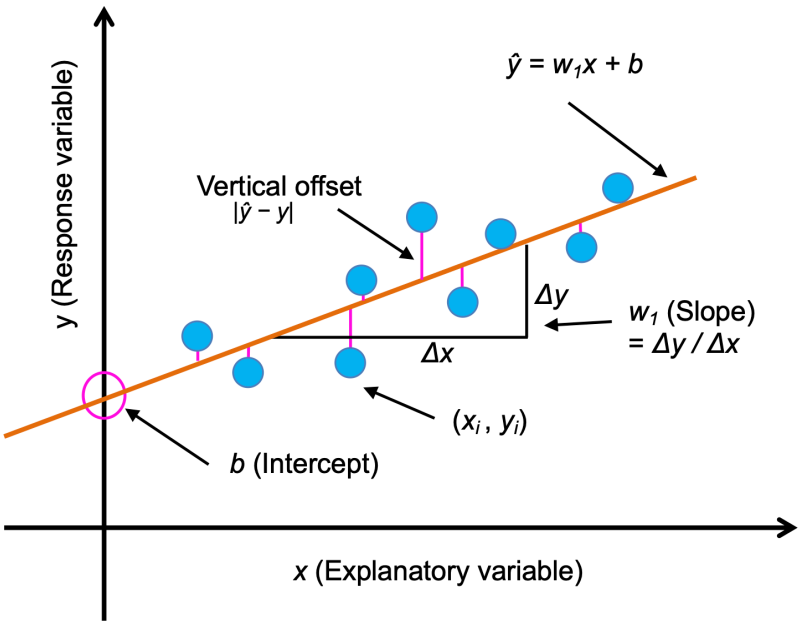

Figure 9.1: A simple one-feature linear regression example

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

Multiple linear regression

The previous section introduced simple linear regression, a special case of linear regression with one explanatory variable. Of course, we can also generalize the linear regression model to multiple explanatory variables; this process is called **multiple linear regression**:

$$y = w_1 x_1 + \ldots + w_m x_m + b = \sum_{i=1}^{m} w_i x_i + b = w^T \boldsymbol{x} + b$$

Figure 9.2 shows how the two-dimensional, fitted hyperplane of a multiple linear regression model with two features could look:

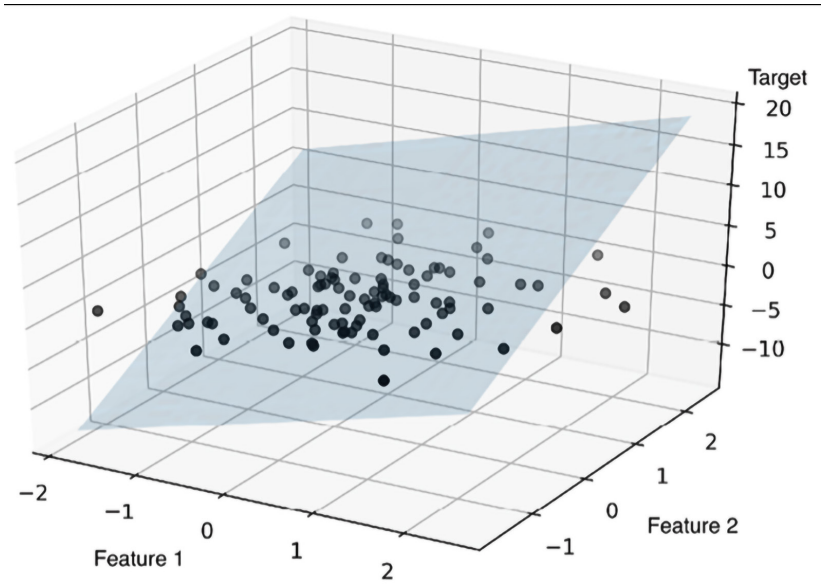

Figure 9.2: A two-feature linear regression model

As you can see, visualizations of multiple linear regression hyperplanes in a three-dimensional scatterplot are already challenging to interpret when looking at static figures. Since we have no good means of visualizing hyperplanes with two dimensions in a scatterplot (multiple linear regression models fit to datasets with three or more features), the examples and visualizations in this chapter will mainly focus on the univariate case, using simple linear regression. However, simple and multiple linear regression are based on the same concepts and the same evaluation techniques; the code implementations that we will discuss in this chapter are also compatible with both types of regression model.

Exploring the Ames Housing dataset

Before we implement the first linear regression model, we will discuss a new dataset, the Ames Housing dataset, which contains information about individual residential property in Ames, Iowa, from 2006 to 2010. The dataset was collected by Dean De Cock in 2011, and additional information is available via the following links:

- A report describing the dataset: `http://jse.amstat.org/v19n3/decock.pdf`
- Detailed documentation regarding the dataset's features: `http://jse.amstat.org/v19n3/decock/DataDocumentation.txt`
- The dataset in a tab-separated format: `http://jse.amstat.org/v19n3/decock/AmesHousing.txt`

As with each new dataset, it is always helpful to explore the data through a simple visualization, to get a better feeling of what we are working with, which is what we will do in the following subsections.

Loading the Ames Housing dataset into a DataFrame

In this section, we will load the Ames Housing dataset using the pandas `read_csv` function, which is fast and versatile and a recommended tool for working with tabular data stored in a plaintext format.

The Ames Housing dataset consists of 2,930 examples and 80 features. For simplicity, we will only work with a subset of the features, shown in the following list. However, if you are curious, follow the link to the full dataset description provided at the beginning of this section, and you are encouraged to explore other variables in this dataset after reading this chapter.

The features we will be working with, including the target variable, are as follows:

- `Overall Qual`: Rating for the overall material and finish of the house on a scale from 1 (very poor) to 10 (excellent)
- `Overall Cond`: Rating for the overall condition of the house on a scale from 1 (very poor) to 10 (excellent)
- `Gr Liv Area`: Above grade (ground) living area in square feet
- `Central Air`: Central air conditioning (N=no, Y=yes)
- `Total Bsmt SF`: Total square feet of the basement area
- `SalePrice`: Sale price in U.S. dollars ($)

For the rest of this chapter, we will regard the sale price (`SalePrice`) as our target variable—the variable that we want to predict using one or more of the five explanatory variables. Before we explore this dataset further, let's load it into a pandas `DataFrame`:

```
import pandas as pd

columns = ['Overall Qual', 'Overall Cond', 'Gr Liv Area',
           'Central Air', 'Total Bsmt SF', 'SalePrice']
```

```
df = pd.read_csv('http://jse.amstat.org/v19n3/decock/AmesHousing.txt',
                 sep='\t',
                 usecols=columns)

df.head()
```

To confirm that the dataset was loaded successfully, we can display the first five lines of the dataset, as shown in *Figure 9.3*:

	Overall Qual	Overall Cond	Total Bsmt SF	Central Air	Gr Liv Area	SalePrice
0	6	5	1080.0	Y	1656	215000
1	5	6	882.0	Y	896	105000
2	6	6	1329.0	Y	1329	172000
3	7	5	2110.0	Y	2110	244000
4	5	5	928.0	Y	1629	189900

Figure 9.3: The first five rows of the housing dataset

After loading the dataset, let's also check the dimensions of the `DataFrame` to make sure that it contains the expected number of rows:

```
>>> df.shape
(2930, 6)
```

As we can see, the `DataFrame` contains 2,930 rows, as expected.

Another aspect we have to take care of is the `'Central Air'` variable, which is encoded as type `string`, as we can see in *Figure 9.3*. As we learned in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, we can use the `.map` method to convert `DataFrame` columns. The following code will convert the string `'Y'` to the integer 1, and the string `'N'` to the integer 0:

```
>>> df['Central Air'] = df['Central Air'].map({'N': 0, 'Y': 1})
```

Lastly, let's check whether any of the data frame columns contain missing values:

```
>>> df.isnull().sum()
Overall Qual     0
Overall Cond     0
Total Bsmt SF    1
Central Air      0
Gr Liv Area      0
SalePrice        0
dtype: int64
```

As we can see, the `Total Bsmt SF` feature variable contains one missing value. Since we have a relatively large dataset, the easiest way to deal with this missing feature value is to remove the corresponding example from the dataset (for alternative methods, please see *Chapter 4*):

```
>>> df = df.dropna(axis=0)
>>> df.isnull().sum()

Overall Qual     0
Overall Cond     0
Total Bsmt SF    0
Central Air      0
Gr Liv Area      0
SalePrice        0
dtype: int64
```

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 `scatterplotmatrix` function from the mlxtend library (`http://rasbt.github.io/mlxtend/`), which is a Python library that contains various convenience functions for machine learning and data science applications in Python.

You can install the `mlxtend` package via `conda install mlxtend` or `pip install mlxtend`. For this chapter, we used mlxtend version 0.19.0.

Once the installation is complete, you can import the package and create the scatterplot matrix as follows:

```
>>> import matplotlib.pyplot as plt
>>> from mlxtend.plotting import scatterplotmatrix
>>> scatterplotmatrix(df.values, figsize=(12, 10),
...                   names=df.columns, alpha=0.5)
>>> plt.tight_layout()
plt.show()
```

As you can see in *Figure 9.4*, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:

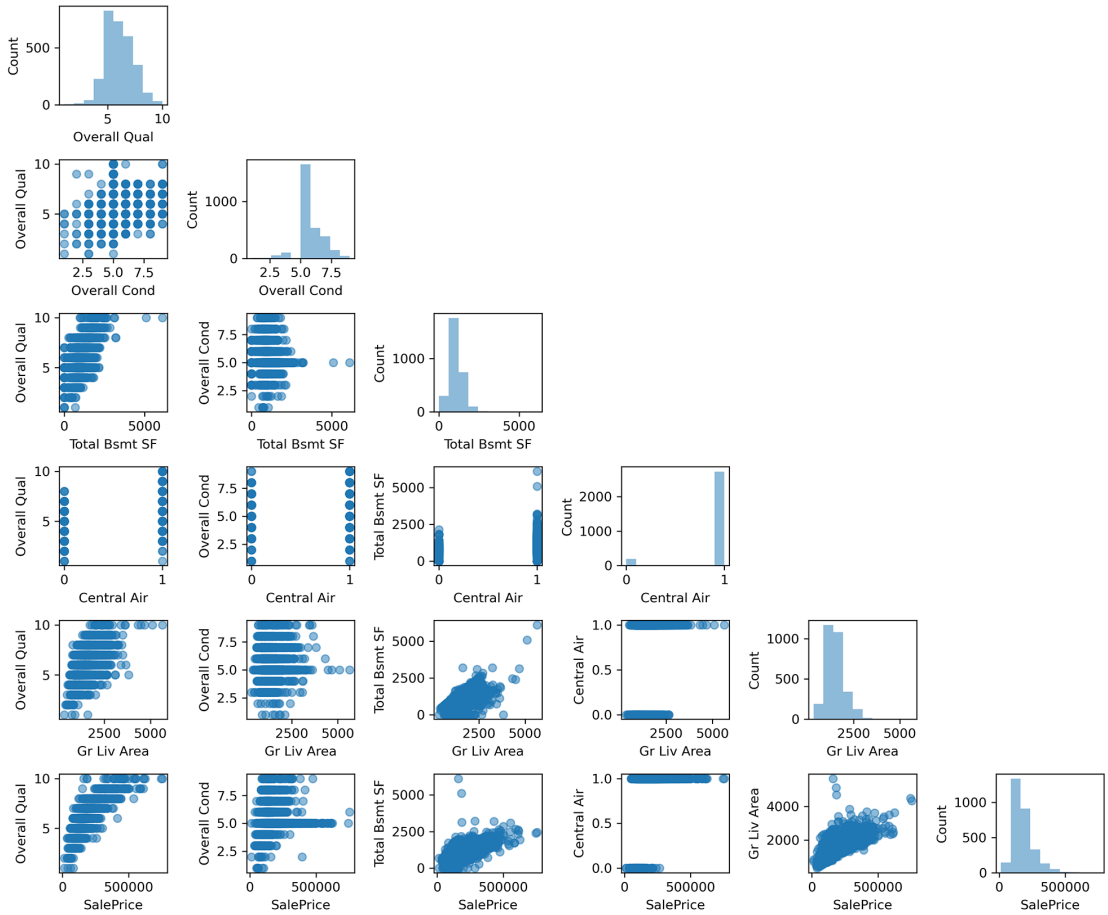

Figure 9.4: A scatterplot matrix of our data

Using this scatterplot matrix, we can now quickly see how the data is distributed and whether it contains outliers. For example, we can see (fifth column from the left of the bottom row) that there is a somewhat linear relationship between the size of the living area above ground (`Gr Liv Area`) and the sale price (`SalePrice`).

Furthermore, we can see in the histogram—the lower-right subplot in the scatterplot matrix—that the `SalePrice` variable seems to be skewed by several outliers.

The normality assumption of linear regression

Note that in contrast to common belief, training a linear regression model does not require that the explanatory or target variables are normally distributed. The normality assumption is only a requirement for certain statistics and hypothesis tests that are beyond the scope of this book (for more information on this topic, please refer to *Introduction to Linear Regression Analysis* by *Douglas C. Montgomery, Elizabeth A. Peck*, and *G. Geoffrey Vining*, *Wiley*, pages: 318-319, 2012).

Looking at relationships using a correlation matrix

In the previous section, we visualized the data distributions of the Ames Housing dataset variables in the form of histograms and scatterplots. 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 covered in the section *Unsupervised dimensionality reduction via principal component analysis* in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. We can interpret the correlation matrix as being 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 measures 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, μ denotes the mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations.

Covariance versus correlation for standardized features

We can show that the covariance between a pair of standardized features is, in fact, equal to their linear correlation coefficient. To show this, let's first standardize the features x and y to obtain their z-scores, which we will denote as x' and y', respectively:

$$x' = \frac{x - \mu_x}{\sigma_x},\ y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we compute the (population) covariance between two features as follows:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Since standardization centers a feature variable at mean zero, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x'^{(i)} - 0\right)\left(y'^{(i)} - 0\right)$$

Through resubstitution, we then get the following result:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\sigma'_{xy} = \frac{1}{n \cdot \sigma_x\sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Finally, we can simplify this equation as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x\sigma_y}$$

In the following code example, we will use NumPy's `corrcoef` function on the five feature columns that we previously visualized in the scatterplot matrix, and we will use mlxtend's `heatmap` function to plot the correlation matrix array as a heat map:

```
>>> import numpy as np
>>> from mlxtend.plotting import heatmap


>>> cm = np.corrcoef(df.values.T)
>>> hm = heatmap(cm, row_names=df.columns, column_names=df.columns)
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in *Figure 9.5*, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective linear correlations:

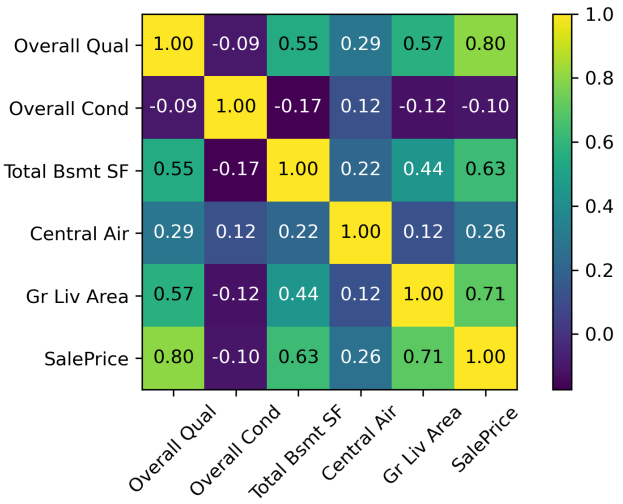

Figure 9.5: A correlation matrix of the selected variables

To fit a linear regression model, we are interested in those features that have a high correlation with our target variable, `SalePrice`. Looking at the previous correlation matrix, we can see that `SalePrice` shows the largest correlation with the `Gr Liv Area` variable (`0.71`), which seems to be a good choice for an exploratory variable to introduce the concepts of a simple linear regression model in the following section.

Implementing an ordinary least squares linear regression model

At the beginning of this chapter, we mentioned that linear regression can be understood as obtaining the best-fitting straight line through the examples of our training data. However, we have neither defined the term *best-fitting* nor have we discussed the different techniques of fitting such a model. In the following subsections, we will fill in the missing pieces of this puzzle using the **ordinary least squares** (**OLS**) method (sometimes also called **linear least squares**) to estimate the parameters of the linear regression line that minimizes the sum of the squared vertical distances (residuals or errors) to the training examples.

Solving regression for regression parameters with gradient descent

Consider our implementation of the **Adaptive Linear Neuron** (**Adaline**) from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. You will remember that the artificial neuron uses a linear activation function. Also, we defined a loss function, $L(\boldsymbol{w})$, which we minimized to learn the weights via optimization algorithms, such as **gradient descent** (**GD**) and **stochastic gradient descent** (**SGD**).

This loss function in Adaline is the **mean squared error** (**MSE**), which is identical to the loss function that we use for OLS:

$$L(\boldsymbol{w}, b) = \frac{1}{2n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

Here, $\hat{y}$ is the predicted value $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$ (note that the term $\frac{1}{2}$ is just used for convenience to derive the update rule of GD). Essentially, OLS regression can be understood as Adaline without the threshold function so that we obtain continuous target values instead of the class labels `0` and `1`. To demonstrate this, let's take the GD implementation of Adaline from *Chapter 2* and remove the threshold function to implement our first linear regression model:

```
class LinearRegressionGD:
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.array([0.])
        self.losses_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        return self.net_input(X)
```

Weight updates with gradient descent

If you need a refresher about how the weights are updated—taking a step in the opposite direction of the gradient—please revisit the *Adaptive linear neurons and the convergence of learning* section in *Chapter 2*.

To see our `LinearRegressionGD` regressor in action, let's use the `Gr Living Area` (size of the living area above ground in square feet) feature from the Ames Housing dataset as the explanatory variable and train a model that can predict `SalePrice`. Furthermore, we will standardize the variables for better convergence of the GD algorithm. The code is as follows:

```
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>>> lr = LinearRegressionGD(eta=0.1)
>>> lr.fit(X_std, y_std)
```

Notice the workaround regarding `y_std`, using `np.newaxis` and `flatten`. Most data preprocessing classes in scikit-learn expect data to be stored in two-dimensional arrays. In the previous code example, the use of `np.newaxis` in `y[:, np.newaxis]` added a new dimension to the array. Then, after `StandardScaler` returned the scaled variable, we converted it back to the original one-dimensional array representation using the `flatten()` method for our convenience.

We discussed in *Chapter 2* that it is always a good idea to plot the loss as a function of the number of epochs (complete iterations) over the training dataset when we are using optimization algorithms, such as GD, to check that the algorithm converged to a loss minimum (here, a *global* loss minimum):

```
>>> plt.plot(range(1, lr.n_iter+1), lr.losses_)
>>> plt.ylabel('MSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

As you can see in *Figure 9.6*, the GD algorithm converged approximately after the tenth epoch:

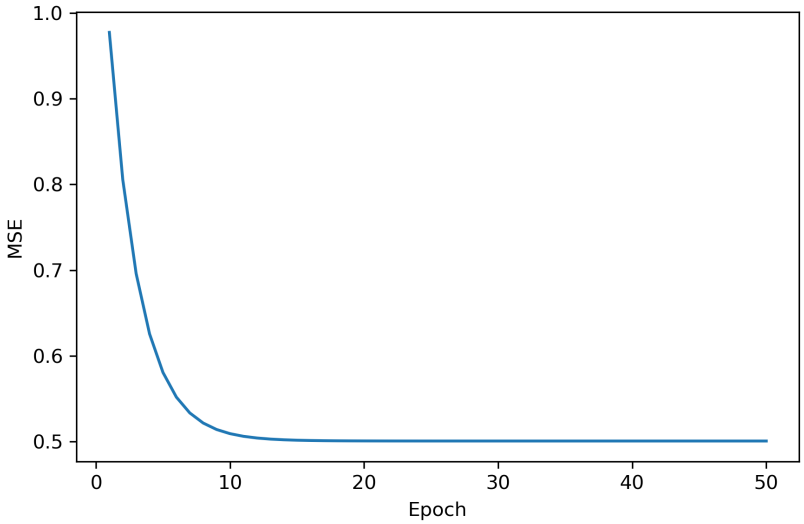

Figure 9.6: The loss function versus the number of epochs

Next, let's visualize how well the linear regression line fits the training data. To do so, we will define a simple helper function that will plot a scatterplot of the training examples and add the regression line:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
...     plt.plot(X, model.predict(X), color='black', lw=2)
```

Now, we will use this `lin_regplot` function to plot the living area against the sale price:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel(' Living area above ground (standardized)')
>>> plt.ylabel('Sale price (standardized)')
>>> plt.show()
```

As you can see in *Figure 9.7*, the linear regression line reflects the general trend that house prices tend to increase with the size of the living area:

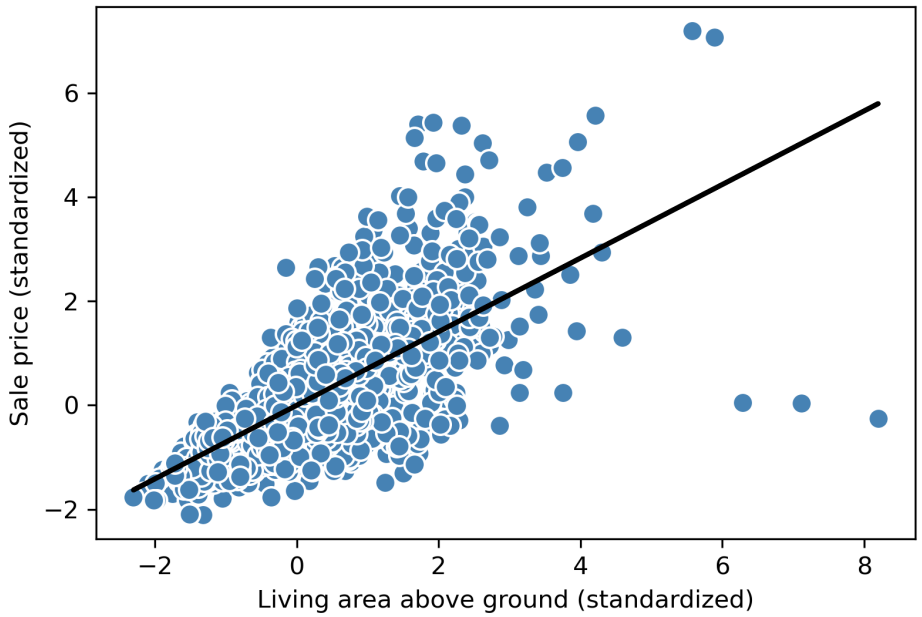

Figure 9.7: A linear regression plot of sale prices versus living area size

Although this observation makes sense, the data also tells us that the living area size does not explain house prices very well in many cases. Later in this chapter, we will discuss how to quantify the performance of a regression model. Interestingly, we can also observe several outliers, for example, the three data points corresponding to a standardized living area greater than 6. We will discuss how we can deal with outliers later in this chapter.

In certain applications, it may also be important to report the predicted outcome variables on their original scale. To scale the predicted price back onto the original *price in U.S. dollars* scale, we can simply apply the `inverse_transform` method of `StandardScaler`:

```
>>> feature_std = sc_x.transform(np.array([[2500]]))
>>> target_std = lr.predict(feature_std)
>>> target_reverted = sc_y.inverse_transform(target_std.reshape(-1, 1))
>>> print(f'Sales price: ${target_reverted.flatten()[0]:.2f}')
Sales price: $292507.07
```

In this code example, we used the previously trained linear regression model to predict the price of a house with an aboveground living area of 2,500 square feet. According to our model, such a house will be worth $292,507.07.

As a side note, it is also worth mentioning that we technically don't have to update the intercept parameter (for instance, the bias unit, *b*) if we are working with standardized variables, since the *y* axis intercept is always 0 in those cases. We can quickly confirm this by printing the model parameters:

```
>>> print(f'Slope: {lr.w_[0]:.3f}')
Slope: 0.707
>>> print(f'Intercept: {lr.b_[0]:.3f}')
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn

In the previous section, we implemented a working model for regression analysis; however, in a real-world application, we may be interested in more efficient implementations. For example, many of scikit-learn's estimators for regression make use of the least squares implementation in SciPy (`scipy.linalg.lstsq`), which, in turn, uses highly optimized code optimizations based on the **Linear Algebra Package** (**LAPACK**). The linear regression implementation in scikit-learn also works (better) with unstandardized variables, since it does not use (S)GD-based optimization, so we can skip the standardization step:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> y_pred = slr.predict(X)
>>> print(f'Slope: {slr.coef_[0]:.3f}')
Slope: 111.666
>>> print(f'Intercept: {slr.intercept_:.3f}')
Intercept: 13342.979
```

As you can see from executing this code, scikit-learn's `LinearRegression` model, fitted with the unstandardized `Gr Liv Area` and `SalePrice` variables, yielded different model coefficients, since the features have not been standardized. However, when we compare it to our GD implementation by plotting `SalePrice` against `Gr Liv Area`, we can qualitatively see that it fits the data similarly well:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')
>>> plt.tight_layout()
>>> plt.show()
```

For instance, we can see that the overall result looks identical to our GD implementation:

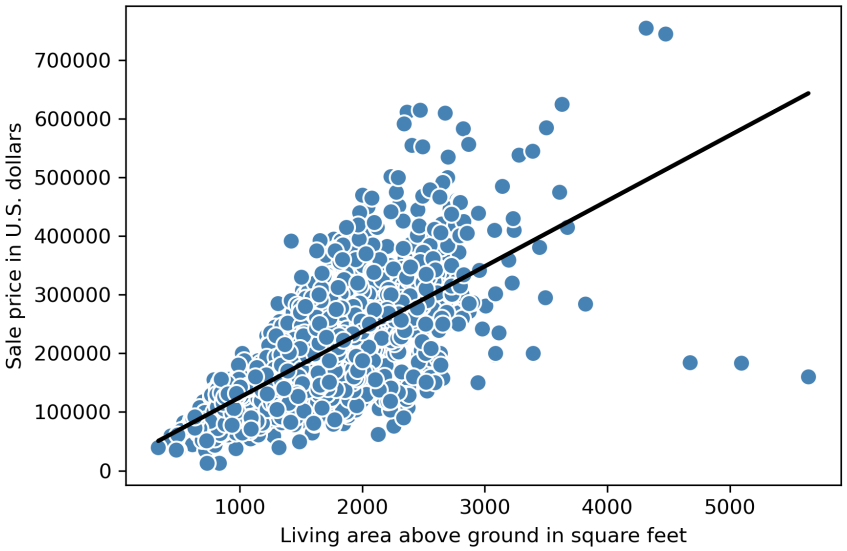

Figure 9.8: A linear regression plot using scikit-learn

Analytical solutions of linear regression

As an alternative to using machine learning libraries, there is also a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$w = (X^T X)^{-1} X^T y$$

We can implement it in Python as follows:

```
# adding a column vector of "ones"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print(f'Slope: {w[1]:.3f}')
Slope: 111.666
>>> print(f'Intercept: {w[0]:.3f}')
Intercept: 13342.979
```

The advantage of this method is that it is guaranteed to find the optimal solution analytically. However, if we are working with very large datasets, it can be computationally too expensive to invert the matrix in this formula (sometimes also called the normal equation), or the matrix containing the training examples may be singular (non-invertible), which is why we may prefer iterative methods in certain cases.

If you are interested in more information on how to obtain normal equations, take a look at Dr. Stephen Pollock's chapter *The Classical Linear Regression Model*, from his lectures at the University of Leicester, which is available for free at `http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf`.

Also, if you want to compare linear regression solutions obtained via GD, SGD, the closed-form solution, QR factorization, and singular vector decomposition, you can use the `LinearRegression` class implemented in mlxtend (`http://rasbt.github.io/mlxtend/user_guide/regressor/LinearRegression/`), which lets users toggle between these options. Another great library to recommend for regression modeling in Python is statsmodels, which implements more advanced linear regression models, as illustrated at `https://www.statsmodels.org/stable/examples/index.html#regression`.

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. Many statistical tests can be used to detect outliers, but these are beyond the scope of the book. However, removing outliers always requires our own judgment as data scientists as well as our domain knowledge.

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 examples 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 was reached; go back to *step 1* otherwise.

Let's now use a linear model in combination with the RANSAC algorithm as implemented in scikit-learn's `RANSACRegressor` class:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(
...     LinearRegression(),
...     max_trials=100, # default value
...     min_samples=0.95,
...     residual_threshold=None, # default value
...     random_state=123)
>>> ransac.fit(X, y)
```

We set the maximum number of iterations of the `RANSACRegressor` to 100, and using `min_samples=0.95`, we set the minimum number of the randomly chosen training examples to be at least 95 percent of the dataset.

By default (via `residual_threshold=None`), 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. You can find a detailed discussion in *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting* by *R. Toldo* and *A. Fusiello*, *Springer*, 2009 (in *Image Analysis and Processing–ICIAP 2009*, pages: 123-131).

Once we have fitted 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 = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='steelblue', edgecolor='white',
...             marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='limegreen', edgecolor='white',
...             marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='black', lw=2)
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in *Figure 9.9*, the linear regression model was fitted on the detected set of inliers, which are shown as circles:

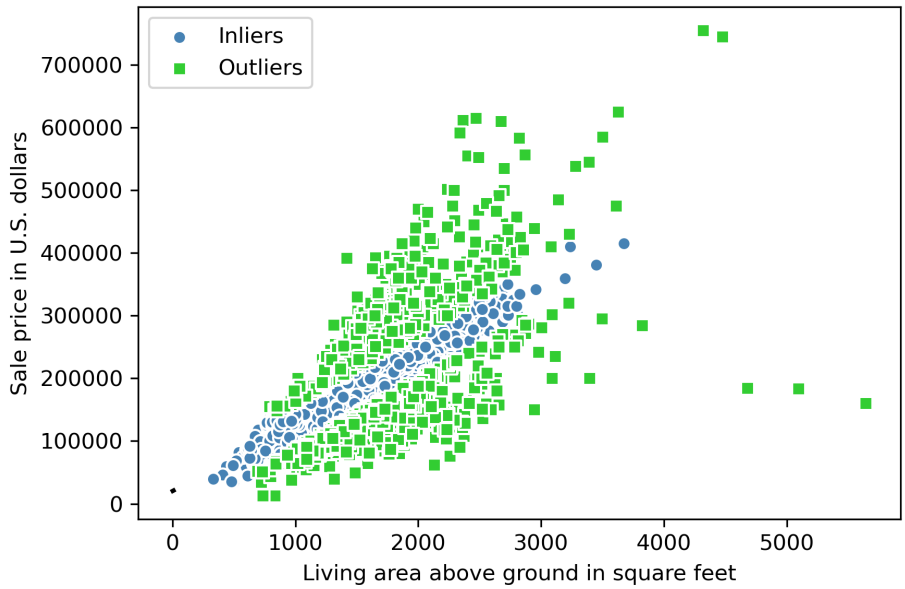

Figure 9.9: Inliers and outliers identified via a RANSAC linear regression model

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

```
>>> print(f'Slope: {ransac.estimator_.coef_[0]:.3f}')
Slope: 106.348
>>> print(f'Intercept: {ransac.estimator_.intercept_:.3f}')
Intercept: 20190.093
```

Remember that we set the `residual_threshold` parameter to `None`, so RANSAC was using the MAD to compute the threshold for flagging inliers and outliers. The MAD, for this dataset, can be computed as follows:

```
>>> def median_absolute_deviation(data):
...     return np.median(np.abs(data - np.median(data)))
>>> median_absolute_deviation(y)
37000.00
```

So, if we want to identify fewer data points as outliers, we can choose a `residual_threshold` value greater than the preceding MAD. For example, *Figure 9.10* shows the inliers and outliers of a RANSAC linear regression model with a residual threshold of 65,000:

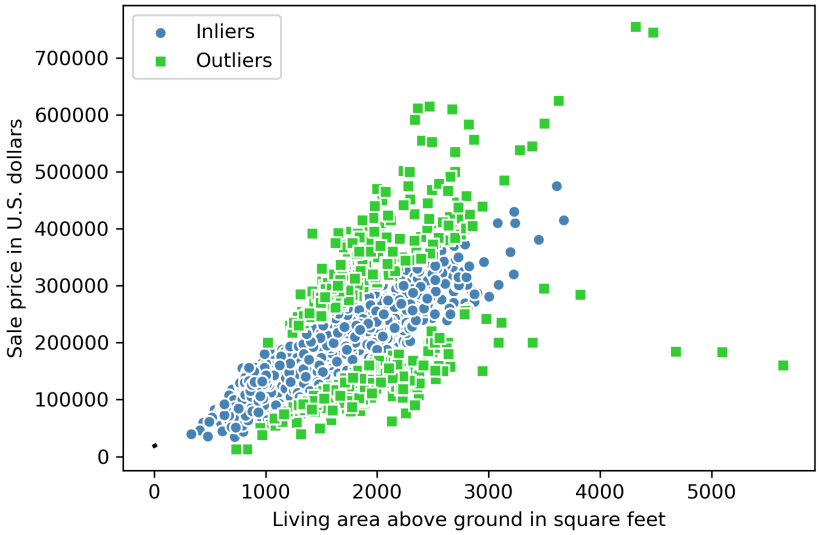

Figure 9.10: Inliers and outliers determined by a RANSAC linear regression model with a larger residual threshold

Using RANSAC, we reduced the potential effect of the outliers in this dataset, but we don't know whether this approach will have a positive effect on the predictive performance for unseen data or not. Thus, in the next section, we will look at different approaches for evaluating a regression model, which is a crucial part of building systems for predictive modeling.

Evaluating the performance of linear regression models

In the previous section, you learned how to fit a regression model on training data. However, you discovered in previous chapters that it is crucial to test the model on data that it hasn't seen during training to obtain a more unbiased estimate of its generalization performance.

As you may remember from *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we want to split our dataset into separate training and test datasets, where we will use the former to fit the model and the latter to evaluate its performance on unseen data to estimate the generalization performance. Instead of proceeding with the simple regression model, we will now use all five features in the dataset and train a multiple regression model:

```
>>> from sklearn.model_selection import train_test_split
>>> target = 'SalePrice'
>>> features = df.columns[df.columns != target]
>>> X = df[features].values
>>> y = df[target].values
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=123)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Since our model uses multiple explanatory variables, we can't visualize the linear regression line (or hyperplane, to be precise) in a two-dimensional plot, but we can plot the residuals (the differences or vertical distances between the actual and predicted values) versus the predicted values to diagnose our regression model. **Residual plots** are a commonly used graphical tool for diagnosing regression models. They can help to detect nonlinearity and outliers and check whether the errors are randomly distributed.

Using the following code, we will now plot a residual plot where we simply subtract the true target variables from our predicted responses:

```
>>> x_max = np.max(
...     [np.max(y_train_pred), np.max(y_test_pred)])
>>> x_min = np.min(
...     [np.min(y_train_pred), np.min(y_test_pred)])

>>> fig, (ax1, ax2) = plt.subplots(
...     1, 2, figsize=(7, 3), sharey=True)

>>> ax1.scatter(
...     y_test_pred, y_test_pred - y_test,
...     c='limegreen', marker='s',
...     edgecolor='white',
...     label='Test data')
>>> ax2.scatter(
...     y_train_pred, y_train_pred - y_train,
...     c='steelblue', marker='o', edgecolor='white',
...     label='Training data')
>>> ax1.set_ylabel('Residuals')

>>> for ax in (ax1, ax2):
...     ax.set_xlabel('Predicted values')
...     ax.legend(loc='upper left')
...     ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,\
...         color='black', lw=2)
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we should see residual plots for the test and training datasets with a line passing through the *x* axis origin, as shown in *Figure 9.11*:

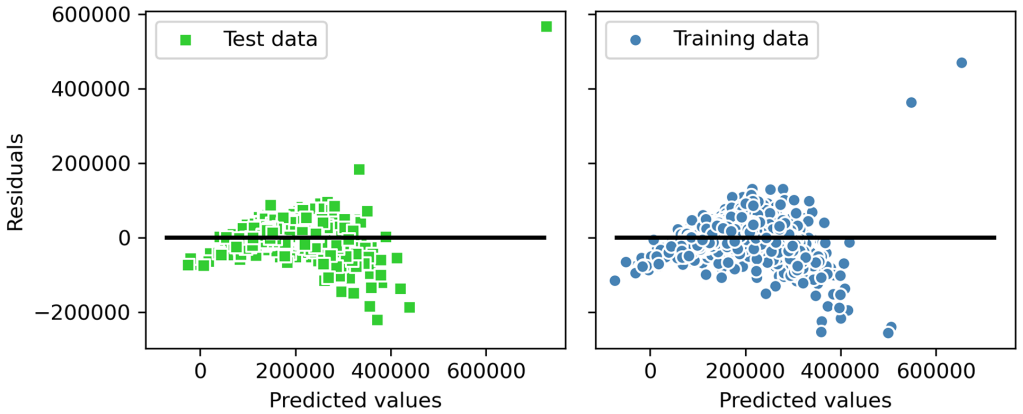

Figure 9.11: Residual plots of our data

In the case of a perfect prediction, the residuals would be exactly zero, which we will probably never encounter in realistic and practical applications. However, for a good regression model, we would expect the errors to be randomly distributed and the residuals to be randomly scattered around the centerline. If we see patterns in a residual plot, it means that our model is unable to capture some explanatory information, which has leaked into the residuals, as you can see to a degree in our previous residual plot. Furthermore, we can also use residual plots to detect outliers, which are represented by the points with a large deviation from the centerline.

Another useful quantitative measure of a model's performance is the **mean squared error** (**MSE**) that we discussed earlier as our loss function that we minimized to fit the linear regression model. The following is a version of the MSE without the $\frac{1}{2}$ scaling factor that is often used to simplify the loss derivative in gradient descent:

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

Similar to prediction accuracy in classification contexts, we can use the MSE for cross-validation and model selection as discussed in *Chapter 6*.

Like classification accuracy, MSE also normalizes according to the sample size, n. This makes it possible to compare across different sample sizes (for example, in the context of learning curves) as well.

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

```
>>> from sklearn.metrics import mean_squared_error
>>> mse_train = mean_squared_error(y_train, y_train_pred)
>>> mse_test = mean_squared_error(y_test, y_test_pred)
```

```
>>> print(f'MSE train: {mse_train:.2f}')
MSE train: 1497216245.85
>>> print(f'MSE test: {mse_test:.2f}')
MSE test: 1516565821.00
```

We can see that the MSE on the training dataset is less than on the test set, which is an indicator that our model is slightly overfitting the training data in this case. Note that it can be more intuitive to show the error on the original unit scale (here, dollar instead of dollar-squared), which is why we may choose to compute the square root of the MSE, called *root mean squared error*, or the **mean absolute error** (**MAE**), which emphasizes incorrect prediction slightly less:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}|$$

We can compute the MAE similar to the MSE:

```
>>> from sklearn.metrics import mean_absolute_error
>>> mae_train = mean_absolute_error(y_train, y_train_pred)
>>> mae_test = mean_absolute_error(y_test, y_test_pred)
>>> print(f'MAE train: {mae_train:.2f}')
MAE train: 25983.03
>>> print(f'MAE test: {mae_test:.2f}')
MAE test: 24921.29
```

Based on the test set MAE, we can say that the model makes an error of approximately $25,000 on average.

When we use the MAE or MSE for comparing models, we need to be aware that these are unbounded in contrast to the classification accuracy, for example. In other words, the interpretations of the MAE and MSE depend on the dataset and feature scaling. For example, if the sale prices were presented as multiples of 1,000 (with the K suffix), the same model would yield a lower MAE compared to a model that worked with unscaled features. To further illustrate this point,

$$|\$500\text{K} - 550\text{K}| < |\$500{,}000 - 550{,}000|$$

Thus, it may sometimes 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, which is similar to the MSE but does not include the normalization by sample size n:

$$SSE = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$$

And SST is the total sum of squares:

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

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

Now, let's briefly show that R^2 is indeed just a rescaled version of the MSE:

$$\begin{aligned} R^2 &= 1 - \frac{\frac{1}{n}SSE}{\frac{1}{n}SST} \\ &= \frac{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \mu_y)^2} \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned}$$

For the training dataset, R^2 is bounded between 0 and 1, but it can become negative for the test dataset. A negative R^2 means that the regression model fits the data worse than a horizontal line representing the sample mean. (In practice, this often happens in the case of extreme overfitting, or if we forget to scale the test set in the same manner we scaled the training set.) If $R^2 = 1$, the model fits the data perfectly with a corresponding $MSE = 0$.

Evaluated on the training data, the R^2 of our model is 0.77, which isn't great but also not too bad given that we only work with a small set of features. However, the R^2 on the test dataset is only slightly smaller, at 0.75, which indicates that the model is only overfitting slightly:

```
>>> from sklearn.metrics import r2_score
>>> train_r2 = r2_score(y_train, y_train_pred)
>>> test_r2 = r2_score(y_test, y_test_pred)
>>> print(f'R^2 train: {train_r2:.3f}, {test_r2:.3f}')
R^2 train: 0.77, test: 0.75
```

Using regularized methods for regression

As we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-Learn*, regularization is one approach to tackling 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 the MSE loss function:

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

Here, the L2 term is defined as follows:

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

By increasing the value of hyperparameter λ, we increase the regularization strength and thereby shrink the weights of our model. Please note that, as mentioned in *Chapter 3*, the bias unit b is not regularized.

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:

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

Here, the L1 penalty for LASSO is defined as the sum of the absolute magnitudes of the model weights, as follows:

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

However, a limitation of LASSO is that it selects at most n features if $m > n$, where n is the number of training examples. This may be undesirable in certain applications of feature selection. In practice, however, this property of LASSO is often an advantage because it avoids saturated models. The saturation of a model occurs if the number of training examples is equal to the number of features, which is a form of overparameterization. As a consequence, a saturated model can always fit the training data perfectly but is merely a form of interpolation and thus is not expected to generalize well.

A compromise between ridge regression and LASSO is elastic net, which has an L1 penalty to generate sparsity and an L2 penalty such that it can be used for selecting more than n features if $m > n$:

$$L(\boldsymbol{w})_{Elastic\ Net} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_2 ||\boldsymbol{w}||_2^2 + \lambda_1 ||\boldsymbol{w}||_1$$

Those regularized regression models are all available via scikit-learn, and their usage is similar to the regular regression model except that we have to specify the regularization strength via the parameter λ, for example, optimized via k-fold cross-validation.

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 λ. 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 `l1_ratio` to 1.0, the `ElasticNet` regressor would be equal to LASSO regression. For more detailed information about the different implementations of linear regression, please refer to the documentation at `http://scikit-learn.org/stable/modules/linear_model.html`.

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:

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

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, $\boldsymbol{w}$. In the following subsections, we will see how we can add such polynomial terms to an existing dataset conveniently and fit a polynomial regression model.

Adding polynomial terms using scikit-learn

We will now learn how to use the `PolynomialFeatures` transformer class from scikit-learn to add a quadratic term ($d = 2$) to a simple regression problem with one explanatory variable. Then, we will compare the polynomial to the linear fit by following these steps:

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 regression model for comparison:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

3. Fit a multiple regression model on the transformed features for polynomial regression:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Plot the results:

```
>>> plt.scatter(X, y, label='Training points')
>>> plt.plot(X_fit, y_lin_fit,
...          label='Linear fit', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...          label='Quadratic fit')
>>> plt.xlabel('Explanatory variable')
>>> plt.ylabel('Predicted or known target values')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

In the resulting plot, you can see that the polynomial fit captures the relationship between the response and explanatory variables much better than the linear fit:

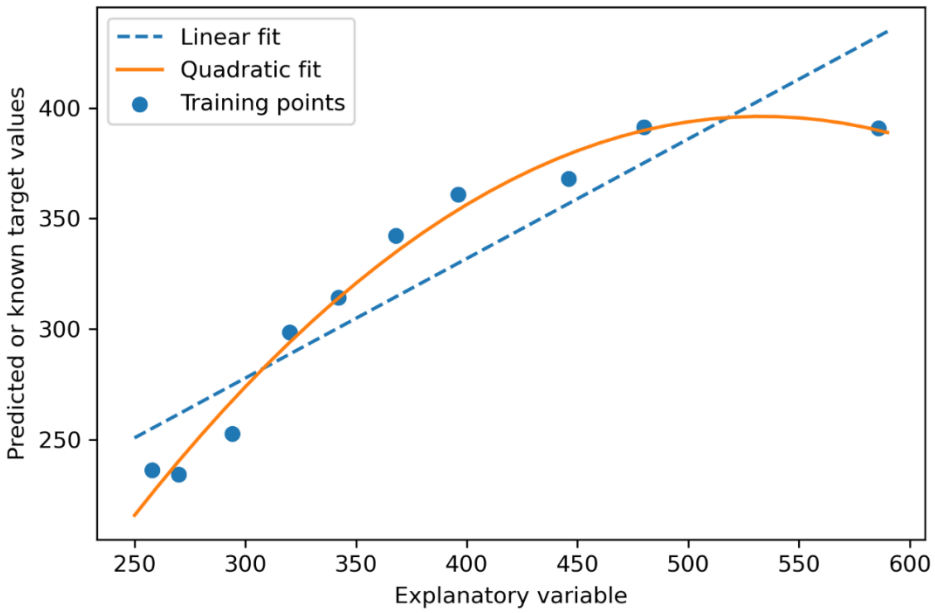

Figure 9.12: A comparison of a linear and quadratic model

Next, we will compute the MSE and R^2 evaluation metrics:

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> mse_lin = mean_squared_error(y, y_lin_pred)
>>> mse_quad = mean_squared_error(y, y_quad_pred)
>>> print(f'Training MSE linear: {mse_lin:.3f}'
          f', quadratic: {mse_quad:.3f}')
Training MSE linear: 569.780, quadratic: 61.330
>>> r2_lin = r2_score(y, y_lin_pred)
>>> r2_quad = r2_score(y, y_quad_pred)
>>> print(f'Training R^2 linear: {r2_lin:.3f}'
          f', quadratic: {r2_quad:.3f}')
Training R^2 linear: 0.832, quadratic: 0.982
```

As you can see after executing the code, the MSE decreased from 570 (linear fit) to 61 (quadratic fit); also, the coefficient of determination reflects a closer fit of the quadratic model ($R^2 = 0.982$) as opposed to the linear fit ($R^2 = 0.832$) in this particular toy problem.

Modeling nonlinear relationships in the Ames Housing dataset

In the preceding subsection, you learned how to construct polynomial features to fit nonlinear relationships in a toy problem; let's now take a look at a more concrete example and apply those concepts to the data in the Ames Housing dataset. By executing the following code, we will model the relationship between sale prices and the living area above ground using second-degree (quadratic) and third-degree (cubic) polynomials and compare that to a linear fit.

We start by removing the three outliers with a living area greater than 4,000 square feet, which we can see in previous figures, such as in *Figure 9.8*, so that these outliers don't skew our regression fits:

```
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> X = X[(df['Gr Liv Area'] < 4000)]
>>> y = y[(df['Gr Liv Area'] < 4000)]
```

Next, we fit the regression models:

```
>>> regr = LinearRegression()

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

>>> # fit to features
>>> X_fit = np.arange(X.min()-1, X.max()+2, 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))

>>> # plot results
>>> plt.scatter(X, y, label='Training points', color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label=f'Linear (d=1), $R^2$={linear_r2:.2f}',
```

```
...           color='blue',
...           lw=2,
...           linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...           label=f'Quadratic (d=2), $R^2$={quadratic_r2:.2f}',
...           color='red',
...           lw=2,
...           linestyle='-')
>>> plt.plot(X_fit, y_cubic_fit,
...           label=f'Cubic (d=3), $R^2$={cubic_r2:.2f}',
...           color='green',
...           lw=2,
...           linestyle='--')
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

The resulting plot is shown in *Figure 9.13*:

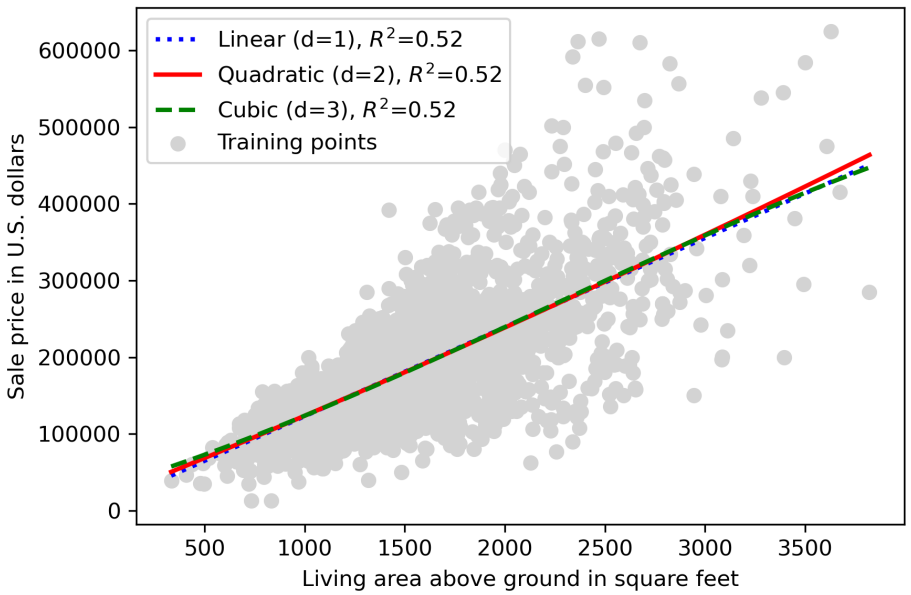

Figure 9.13: A comparison of different curves fitted to the sale price and living area data

As we can see, using quadratic or cubic features does not really have an effect. That's because the relationship between the two variables appears to be linear. So, let's take a look at another feature, namely, `Overall Qual`. The `Overall Qual` variable rates the overall quality of the material and finish of the houses and is given on a scale from 1 to 10, where 10 is best:

```
>>> X = df[['Overall Qual']].values
>>> y = df['SalePrice'].values
```

After specifying the `X` and `y` variables, we can reuse the previous code and obtain the plot in *Figure 9.14*:

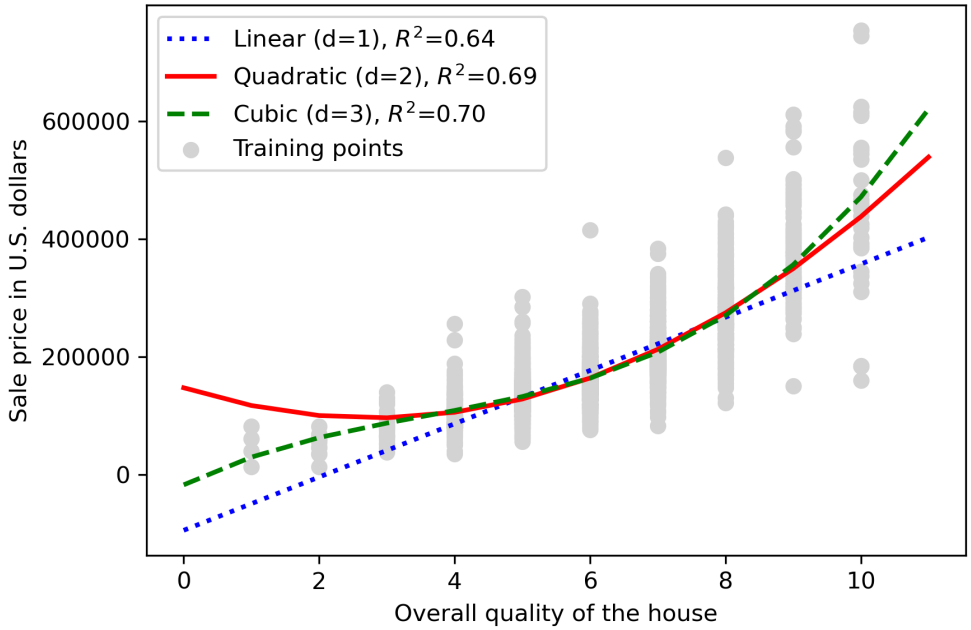

Figure 9.14: A linear, quadratic, and cubic fit on the sale price and house quality data

As you can see, the quadratic and cubic fits capture the relationship between sale prices and the overall quality of the house better than the linear fit. However, you should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting. Thus, in practice, it is always recommended to evaluate the performance of the model on a separate test dataset to estimate the generalization performance.

Dealing with nonlinear relationships using random forests

In this section, we are going to 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 subdivide the input space into smaller regions that become more manageable.

Decision tree regression

An advantage of the decision tree algorithm is that it works with arbitrary features and does not require any transformation of the features if we are dealing with nonlinear data because decision trees analyze one feature at a time, rather than taking weighted combinations into account. (Likewise, normalizing or standardizing features is not required for decision trees.) As mentioned in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, we grow a 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:

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

Here, x_i is the feature to perform the split, N_p is the number of training examples in the parent node, I is the impurity function, D_p is the subset of training examples at the parent node, and D_{left} and D_{right} are the subsets of training examples at the left and right child nodes after the split. Remember that our goal is to find the feature split that maximizes the information gain; in other words, we want to find the feature split that reduces the impurities in the child nodes most. In *Chapter 3*, we discussed Gini impurity and entropy as measures of impurity, which are both useful criteria for classification. To use a decision tree for regression, however, we need an impurity metric that is suitable for continuous variables, so we define the impurity measure of a node, t, as the MSE instead:

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

Here, N_t is the number of training examples 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):

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

In the context of decision tree regression, the MSE is often referred to as **within-node variance**, which is why the splitting criterion is also better known as **variance reduction**.

To see what the line fit of a decision tree looks like, let's use the `DecisionTreeRegressor` implemented in scikit-learn to model the relationship between the `SalePrice` and `Gr Living Area` variables. Note that `SalePrice` and `Gr Living Area` do not necessarily represent a nonlinear relationship, but this feature combination still demonstrates the general aspects of a regression tree quite nicely:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
```

```
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')>>> plt.show()
```

As you can see in the resulting plot, the decision tree captures the general trend in the data. And we can imagine that a regression tree could also capture trends in nonlinear data relatively well. 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 so as to not overfit or underfit the data; here, a depth of three seemed to be a good choice.

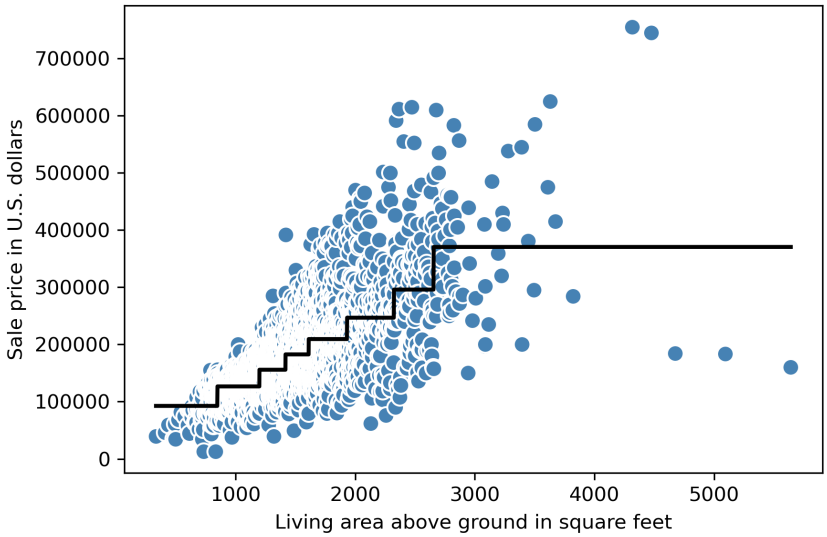

Figure 9.15: A decision tree regression plot

You are encouraged to experiment with deeper decision trees. Note that the relationship between `Gr Living Area` and `SalePrice` is rather linear, so you are also encouraged to apply the decision tree to the `Overall Qual` variable instead.

In the next section, we will look at a more robust way of fitting regression trees: random forests.

Random forest regression

As you learned in *Chapter 3*, the 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*. 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 across all decision trees.

Now, let's use all the features in the Ames Housing dataset to fit a random forest regression model on 70 percent of the examples and evaluate its performance on the remaining 30 percent, as we have done previously in the *Evaluating the performance of linear regression models* section. The code is as follows:

```
>>> target = 'SalePrice'
>>> features = df.columns[df.columns != target]
>>> X = df[features].values
>>> y = df[target].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=123)

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(
...     n_estimators=1000,
...     criterion='squared_error',
...     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)
>>> mae_train = mean_absolute_error(y_train, y_train_pred)
>>> mae_test = mean_absolute_error(y_test, y_test_pred)
>>> print(f'MAE train: {mae_train:.2f}')
MAE train: 8305.18
>>> print(f'MAE test: {mae_test:.2f}')
MAE test: 20821.77
>>> r2_train = r2_score(y_train, y_train_pred)
>>> r2_test =r2_score(y_test, y_test_pred)
>>> print(f'R^2 train: {r2_train:.2f}')
R^2 train: 0.98
>>> print(f'R^2 test: {r2_test:.2f}')
R^2 test: 0.85
```

Unfortunately, you can see that the random forest tends to overfit the training data. However, it's still able to explain the relationship between the target and explanatory variables relatively well ($R^2 = 0.85$ on the test dataset). For comparison, the linear model from the previous section, *Evaluating the performance of linear regression models*, which was fit to the same dataset, was overfitting less but performed worse on the test set ($R^2 = 0.75$).

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

```
>>> x_max = np.max([np.max(y_train_pred), np.max(y_test_pred)])
>>> x_min = np.min([np.min(y_train_pred), np.min(y_test_pred)])

>>> fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(7, 3), sharey=True)
```

```
>>> ax1.scatter(y_test_pred, y_test_pred - y_test,
...             c='limegreen', marker='s', edgecolor='white',
...             label='Test data')
>>> ax2.scatter(y_train_pred, y_train_pred - y_train,
...             c='steelblue', marker='o', edgecolor='white',
...             label='Training data')
>>> ax1.set_ylabel('Residuals')

>>> for ax in (ax1, ax2):
...     ax.set_xlabel('Predicted values')
...     ax.legend(loc='upper left')
...     ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,
...               color='black', lw=2)

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

As it was already summarized by the R^2 coefficient, you can see that the model fits the training data better than the test data, as indicated by the outliers in the y axis direction. Also, the distribution of the residuals does not seem to be completely random around the zero center point, indicating that the model is not able to capture all the exploratory information. However, the residual plot indicates a large improvement over the residual plot of the linear model that we plotted earlier in this chapter.

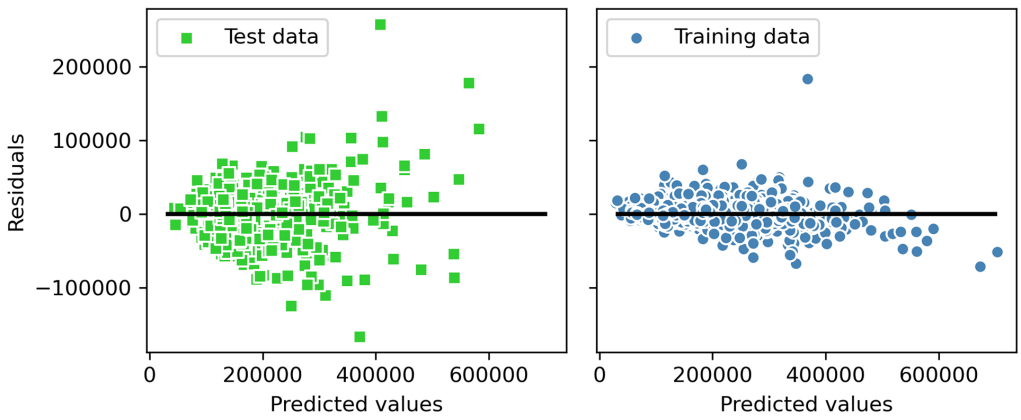

Figure 9.16: The residuals of the random forest regression

Ideally, our model error should be random or unpredictable. In other words, the error of the predictions should not be related to any of the information contained in the explanatory variables; rather, it should reflect the randomness of the real-world distributions or patterns. If we find patterns in the prediction errors, for example, by inspecting the residual plot, it means that the residual plots contain predictive information. A common reason for this could be that explanatory information is leaking into those residuals.

Unfortunately, there is not a universal approach for dealing with non-randomness in residual plots, and it requires experimentation. Depending on the data that is available to us, we may be able to improve the model by transforming variables, tuning the hyperparameters of the learning algorithm, choosing simpler or more complex models, removing outliers, or including additional variables.

Summary

At the beginning of this chapter, you learned about simple linear regression analysis to model the relationship between a single explanatory variable and a continuous response variable. We then discussed a useful explanatory data analysis technique to look at patterns and anomalies in data, which is an important first step in predictive modeling tasks.

We built our first model by implementing linear regression using a gradient-based optimization approach. You then saw how to utilize scikit-learn's linear models for regression and also implement a robust regression technique (RANSAC) as an approach for dealing with outliers. To assess the predictive performance of regression models, we computed the mean sum of squared errors and the related R^2 metric. Furthermore, we also discussed a useful graphical approach for diagnosing the problems of regression models: the residual plot.

After we explored how regularization can be applied to regression models to reduce the model complexity and avoid overfitting, we also covered several approaches for modeling nonlinear relationships, including polynomial feature transformation and random forest regressors.

We discussed supervised learning, classification, and regression analysis in detail in the previous chapters. In the next chapter, we are going to learn about another interesting subfield of machine learning, unsupervised learning, and also how to use cluster analysis to find hidden structures in data in the absence of target variables.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`