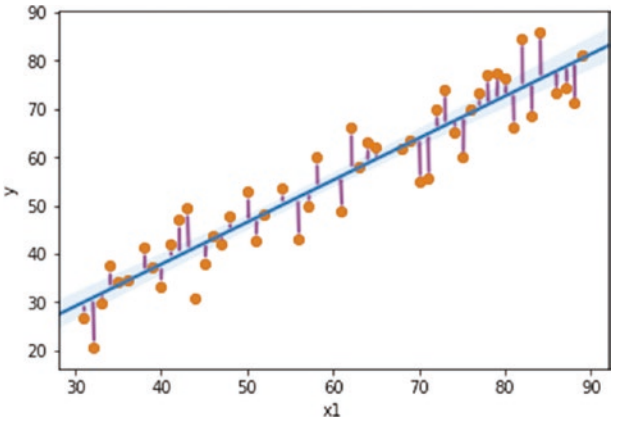


Figure 19-4. *Linear regression model showing residuals*

If all the points in Figure 19-4 entirely fall on the predicted regression line, then the error will be 0. In interpreting the regression model, we want the error measure to be as low as possible.

However, our emphasis is to obtain a low error measure when we evaluate our model on the test dataset. Recall that the test of learning is when a model can generalize to examples that it was not exposed to during training.

Linear Regression with Scikit-learn

In this example, we will implement a linear regression model with Scikit-learn. The model will predict house prices from the Boston house-prices dataset. The dataset contains 506 observations and 13 features.

We begin by importing the following packages:

```
sklearn.linear_model.LinearRegression: function that implements the
LinearRegression model.
sklearn.datasets: function to load sample datasets integrated with scikit-
learn for experimental and learning purposes.
sklearn.model_selection.train_test_split: function that partitions the
dataset into train and test splits.
sklearn.metrics.mean_squared_error: function to load the evaluation metric
for checking the performance of the model.
```

```
math.sqrt: imports the square-root math function. It is used later to
calculate the RMSE when evaluating the model.
# import packages
from sklearn.linear_model import LinearRegression
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from math import sqrt

# load dataset
data = datasets.load_boston()
# separate features and target
X = data.data
y = data.target

# split in train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True)

# create the model
# setting normalize to true normalizes the dataset before fitting the model
linear_reg = LinearRegression(normalize = True)

# fit the model on the training set
linear_reg.fit(X_train, y_train)
'Output': LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=True)

# make predictions on the test set
predictions = linear_reg.predict(X_test)

# evaluate the model performance using the root mean square error metric
print("Root mean squared error (RMSE): %.2f" % sqrt(mean_squared_error(y_
test, predictions)))
'Output':
Root mean squared error (RMSE): 4.33
```

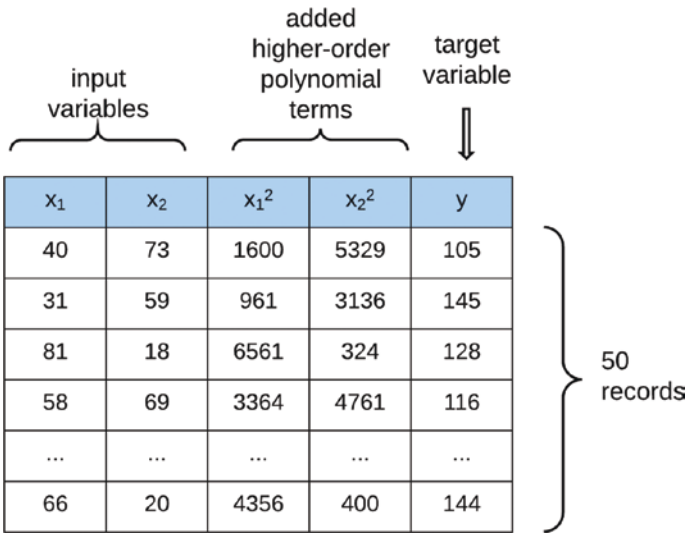
In the preceding code, using the *train_test_split()* function, the dataset is split into training and testing sets. The linear regression algorithm is applied to the training dataset to find the optimal values that parameterize the weights of the model. The model is evaluated by calling the *.predict()* function on the test set.

The error of the model is evaluated using the RMSE error metric (discussed in Chapter 14).

Adapting to Non-linearity

Although linear regression has the premise that the underlying structure of the dataset features is linear, this is, however, not the case for most datasets. It is nevertheless possible to adapt linear regression to fit or build a model for non-linear datasets. This process of adding non-linearity to linear models is called ***polynomial regression***.

Polynomial regression fits a non-linear relationship to the data by adding higher-order polynomial terms of existing data features as new features in the dataset. More of this is visualized in Figure 19-5.



x_1	x_2	x_1^2	x_2^2	y
40	73	1600	5329	105
31	59	961	3136	145
81	18	6561	324	128
58	69	3364	4761	116
...	...	...	...	...
66	20	4356	400	144

Figure 19-5. *Adding polynomial features to the dataset*