It is important to note that from a statistical point of view, when approximating the optimal values of the weights to minimize the model, the underlying assumption of the interactions of the parameters is linear. Non-linear regression models may tend to overfit the data, but this can be mitigated by adding regularization to the model. Here is a formal example of the polynomial regression model.

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_2 + \theta_4 x_2^2 + \ldots + \theta_n x_n + \theta_n x_n^2$$

An illustration of polynomial regression is shown in Figure 19-6.

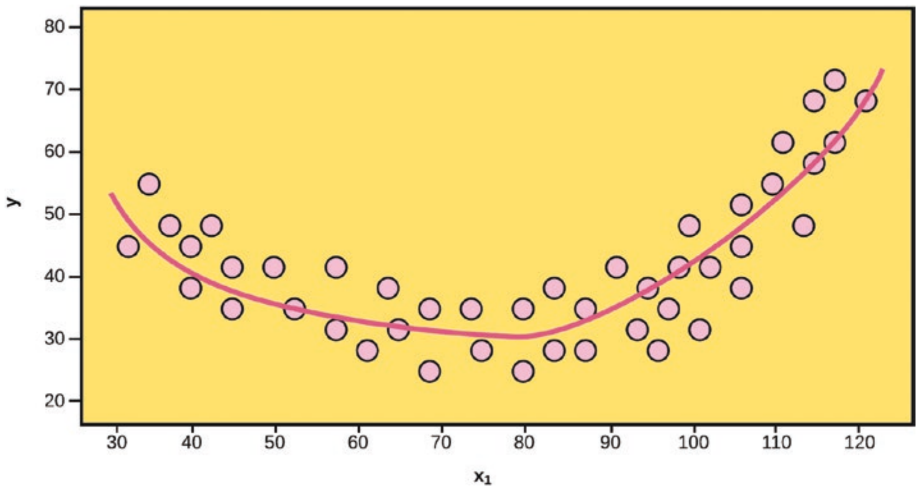

***Figure 19-6.** Fitting a non-linear model with polynomial regression*

Higher-Order Linear Regression with Scikit-learn

In this example, we will create higher-order polynomials from the dataset features in hope of fitting a more flexible model that may better capture the variance in the dataset. As seen in Chapter 18, we will use the PolynomialFeatures method to create these higher-order polynomial and interaction features. The following code example is similar to the previous code example except where it extends the feature matrix with higher-order features.

```
# 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
from sklearn.preprocessing import PolynomialFeatures

# load dataset
data = datasets.load_boston()

# separate features and target
X = data.data
y = data.target

# create polynomial features
polynomial_features = PolynomialFeatures(2)
X_higher_order = polynomial_features.fit_transform(X)

# split in train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_higher_order, 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=None,
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): 3.01
```

From the example, we can observe a slight improvement in the error score of the model with added higher-order features. This result is similar to what may most likely be observed in practice. It is rare to find datasets from real-world events where the features have a perfectly underlying linear structure. So adding higher-order terms is most likely to improve the model performance. But we must watch out to avoid overfitting the model.

Improving the Performance of a Linear Regression Model

The following techniques are options that can be explored to improve the performance of a linear regression model.

In the case of Bias (i.e., poor MSE on training data)

- Perform feature selection to reduce the parameter space. Feature selection is the process of eliminating variables that do not contribute to learning the prediction model. There are various automatic methods for feature selection with linear regression. A couple of them are backward selection, forward propagation, and stepwise regression. Features can also be pruned manually by systematically going through each feature in the dataset and determining its relevance to the learning problem.
- Remove features with high correlation. Correlation occurs when two predictor features are strongly dependent on one another. Empirically, highly correlated features in the datasets may hurt the model accuracy.
- Use higher-order features. A more flexible fit may better capture the variance in the dataset.
- Rescale your data before training. Unscaled features negatively affect the prediction quality of a regression model. Because of the different feature scales in multi-dimensional space, it becomes difficult for the model to find the optimal weights that capture the learning problem. As mentioned in Chapter 16, gradient descent performs better with feature scaling.