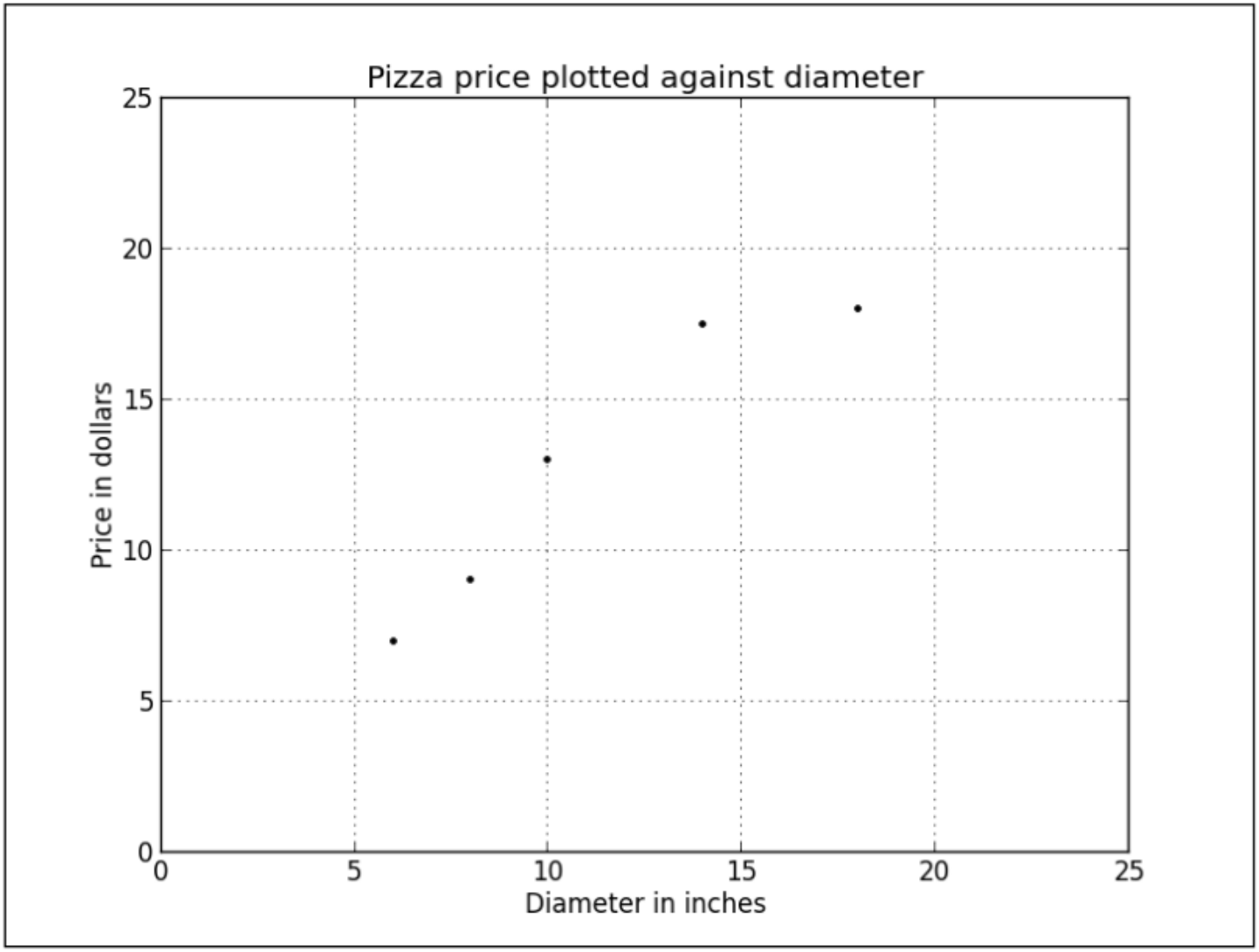
Suppose you wish to know the price of a pizza. You might simply look at a menu. This, however, is a machine learning book, so we will use simple linear regression instead to predict the price of a pizza based on an attribute of the pizza that we can observe. Let's model the relationship between the size of a pizza and its price. First, we will write a program with scikit-learn that can predict the price of a pizza given its size. Then, we will discuss how simple linear regression works and how it can be generalized to work with other types of problems. Let's assume that you have recorded the diameters and prices of pizzas that you have previously eaten in your pizza journal. These observations comprise our training data:

Training instance	**Diameter (in inches)**	**Price (in dollars)**
1	6	7
2	8	9
3	10	13
4	14	17.5
5	18	18

We can visualize our training data by plotting it on a graph using `matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> X = [[6], [8], [10], [14],   [18]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> plt.figure()
>>> plt.title('Pizza price plotted against diameter')
>>> plt.xlabel('Diameter in inches')
>>> plt.ylabel('Price in dollars')
>>> plt.plot(X, y, 'k.')
>>> plt.axis([0, 25, 0, 25])
>>> plt.grid(True)
>>> plt.show()
```

The preceding script produces the following graph. The diameters of the pizzas are plotted on the *x* axis and the prices are plotted on the *y* axis.



We can see from the graph of the training data that there is a positive relationship between the diameter of a pizza and its price, which should be corroborated by our own pizza-eating experience. As the diameter of a pizza increases, its price generally increases too. The following pizza-price predictor program models this relationship using linear regression. Let's review the following program and discuss how linear regression works:

```
>>> from sklearn.linear_model import LinearRegression
>>> # Training data
>>> X = [[6], [8], [10], [14],   [18]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> # Create and fit the model
>>> model = LinearRegression()
>>> model.fit(X, y)
>>> print 'A 12" pizza should cost: $%.2f' % model.predict([12])[0]
A 12" pizza should cost: $13.68
```

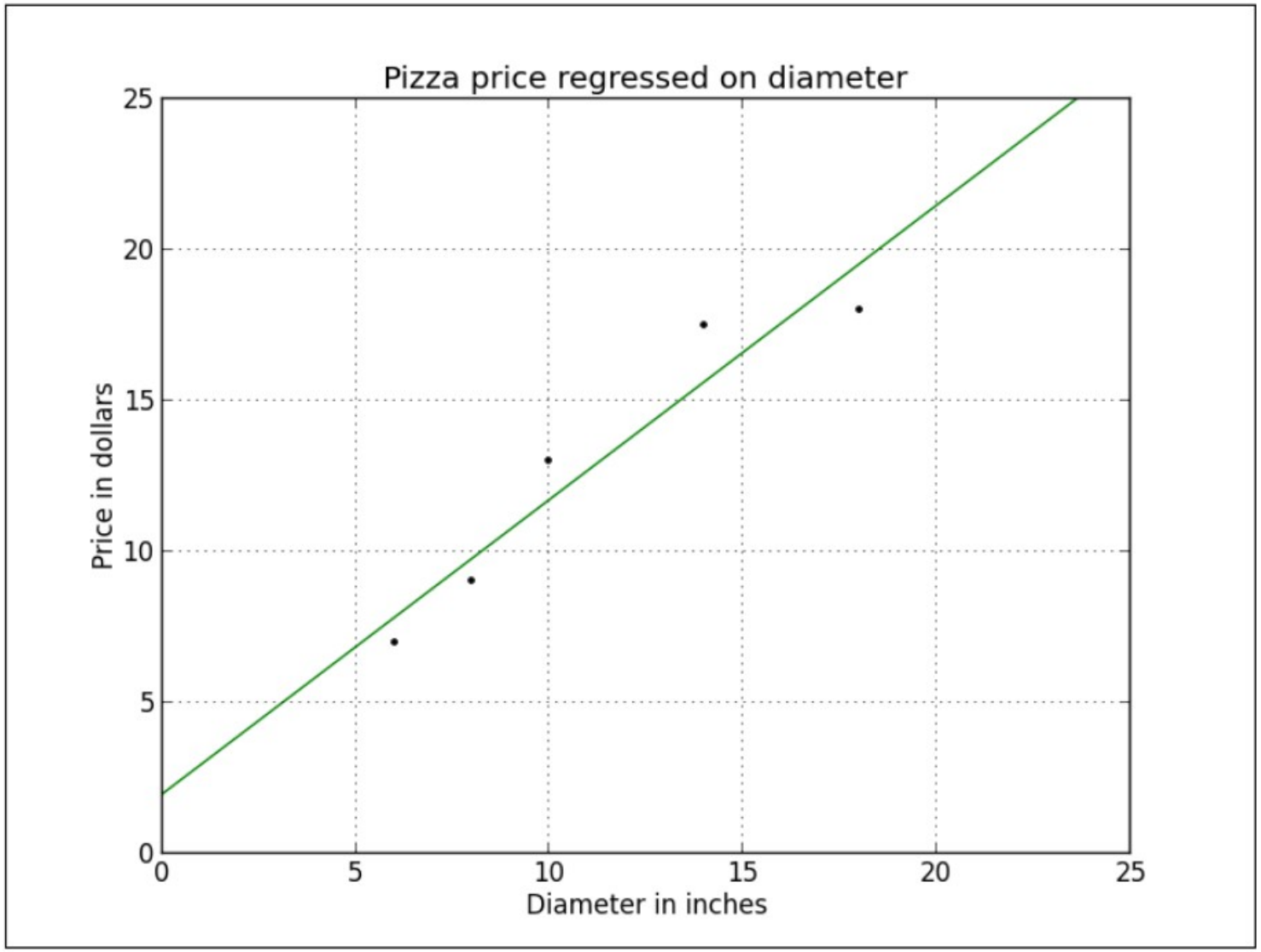
Simple linear regression assumes that a linear relationship exists between the response variable and explanatory variable; it models this relationship with a linear surface called a hyperplane. A hyperplane is a subspace that has one dimension less than the ambient space that contains it. In simple linear regression, there is one dimension for the response variable and another dimension for the explanatory variable, making a total of two dimensions. The regression hyperplane therefore, has one dimension; a hyperplane with one dimension is a line.

The `sklearn.linear_model.LinearRegression` class is an **estimator**. Estimators predict a value based on the observed data. In scikit-learn, all estimators implement the `fit()` and `predict()` methods. The former method is used to learn the parameters of a model, and the latter method is used to predict the value of a response variable for an explanatory variable using the learned parameters. It is easy to experiment with different models using scikit-learn because all estimators implement the `fit` and `predict` methods.

The `fit` method of `LinearRegression` learns the parameters of the following model for simple linear regression:

$$y = \alpha + \beta x$$

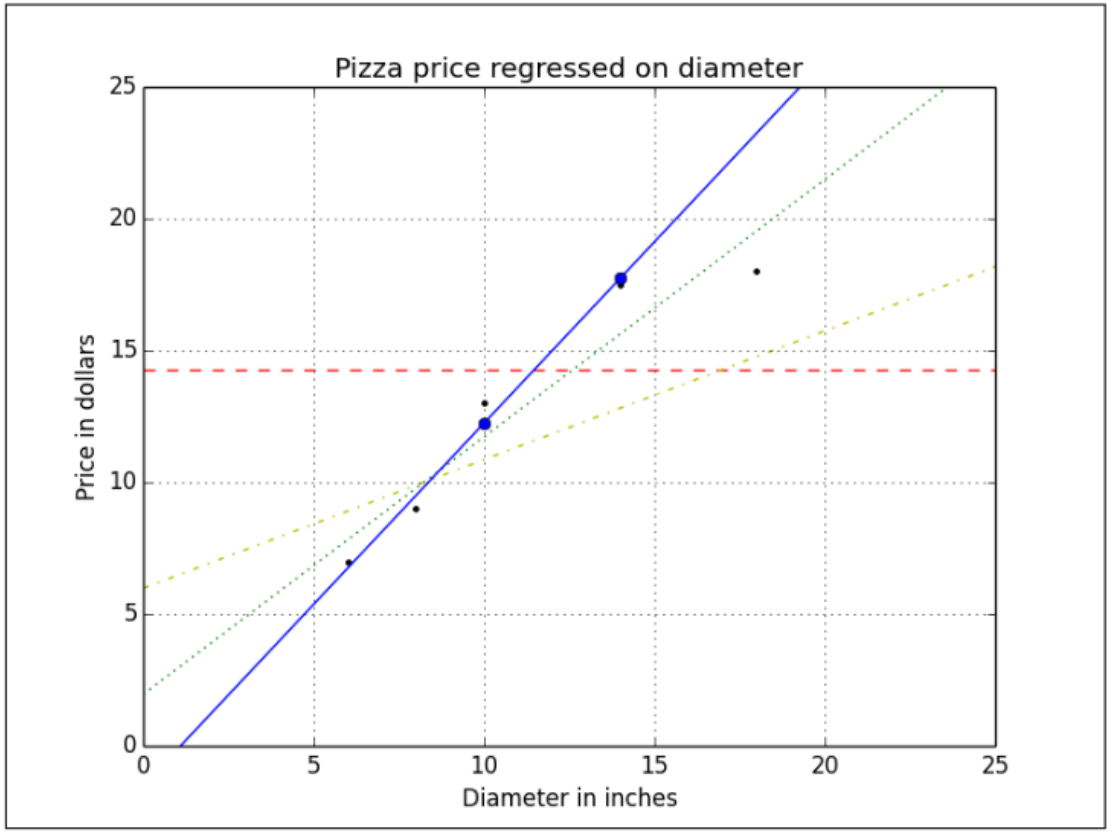
y is the predicted value of the response variable; in this example, it is the predicted price of the pizza. x is the explanatory variable. The intercept term α and coefficient β are parameters of the model that are learned by the learning algorithm. The line plotted in the following figure models the relationship between the size of a pizza and its price. Using this model, we would expect the price of an 8-inch pizza to be about $7.33, and the price of a 20-inch pizza to be $18.75.



Using training data to learn the values of the parameters for simple linear regression that produce the best fitting model is called **ordinary least squares** or **linear least squares**. "In this chapter we will discuss methods for approximating the values of the model's parameters and for solving them analytically. First, however, we must define what it means for a model to fit the training data.

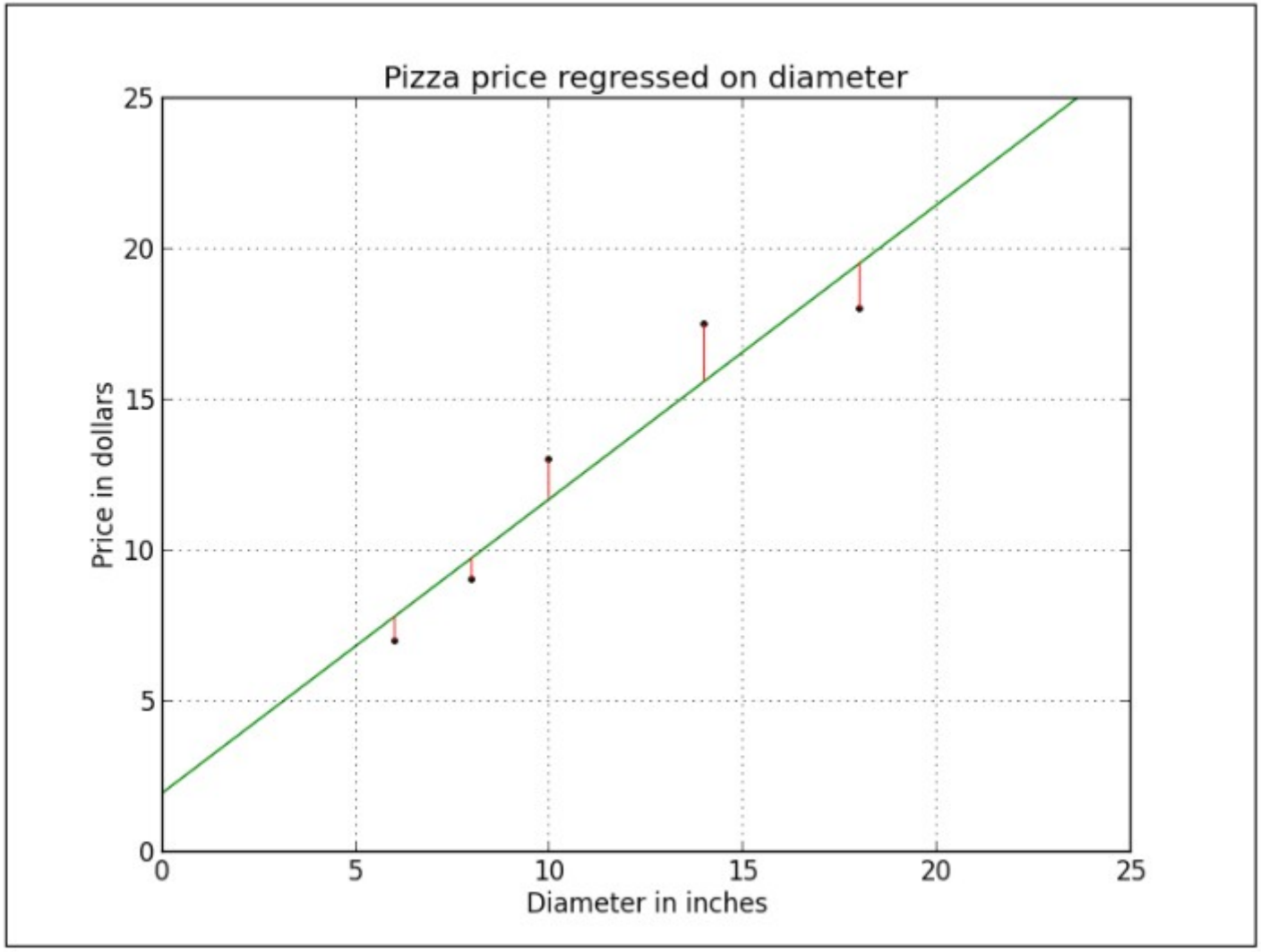
Evaluating the fitness of a model with a cost function

Regression lines produced by several sets of parameter values are plotted in the following figure. How can we assess which parameters produced the best-fitting regression line?



A **cost function**, also called a **loss function,** is used to define and measure the error of a model. The differences between the prices predicted by the model and the observed prices of the pizzas in the training set are called **residuals** or **training errors**. Later, we will evaluate a model on a separate set of test data; the differences between the predicted and observed values in the test data are called **prediction errors** or **test errors**.

The residuals for our model are indicated by the vertical lines between the points for the training instances and regression hyperplane in the following plot:



We can produce the best pizza-price predictor by minimizing the sum of the residuals. That is, our model fits if the values it predicts for the response variable are close to the observed values for all of the training examples. This measure of the model's fitness is called the **residual sum of squares** cost function. Formally, this function assesses the fitness of a model by summing the squared residuals for all of our training examples. The residual sum of squares is calculated with the formula in the following equation, where y_i is the observed value and $f(x_i)$ is the predicted value:

$$SS_{res} = \sum_{i=1}^{n} \left(y_i - f(x_i) \right)^2$$

Let's compute the residual sum of squares for our model by adding the following two lines to the previous script:

```
>>> import numpy as np
>>> print 'Residual sum of squares: %.2f' % np.mean((model.predict(X)
- y) ** 2)
Residual sum of squares: 1.75
```

Now that we have a cost function, we can find the values of our model's parameters that minimize it.

Solving ordinary least squares for simple linear regression

In this section, we will work through solving ordinary least squares for simple linear regression. Recall that simple linear regression is given by the following equation:

$$y = \alpha + \beta x$$

Also, recall that our goal is to solve the values of β and α that minimize the cost function. We will solve β first. To do so, we will calculate the **variance** of x and **covariance** of x and y.

Variance is a measure of how far a set of values is spread out. If all of the numbers in the set are equal, the variance of the set is zero. A small variance indicates that the numbers are near the mean of the set, while a set containing numbers that are far from the mean and each other will have a large variance. Variance can be calculated using the following equation:

$$\text{var}(x) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}$$

In the preceding equation, $\bar{x}$ is the mean of x, x_i is the value of x for the *ith* training instance, and n is the number of training instances. Let's calculate the variance of the pizza diameters in our training set:

```
>>> from __future__ import division
>>> xbar = (6 + 8 + 10 + 14 + 18) / 5
>>> variance = ((6 - xbar)**2 + (8 - xbar)**2 + (10 - xbar)**2 + (14 -
xbar)**2 + (18 - xbar)**2) / 4
>>> print variance
23.2
```

NumPy also provides the `var` method to calculate variance. The `ddof` keyword parameter can be used to set Bessel's correction to calculate the sample variance:

```
>>> import numpy as np
>>> print np.var([6, 8, 10, 14, 18], ddof=1)
23.2
```

Covariance is a measure of how much two variables change together. If the value of the variables increase together, their covariance is positive. If one variable tends to increase while the other decreases, their covariance is negative. If there is no linear relationship between the two variables, their covariance will be equal to zero; the variables are linearly uncorrelated but not necessarily independent. Covariance can be calculated using the following formula:

$$\text{cov}(x, y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n-1}$$

As with variance, x_i is the diameter of the ith training instance, $\bar{x}$ is the mean of the diameters, $\bar{y}$ is the mean of the prices, y_i is the price of the ith training instance, and n is the number of training instances. Let's calculate the covariance of the diameters and prices of the pizzas in the training set:

```
>>> xbar = (6 + 8 + 10 + 14 + 18) / 5
>>> ybar = (7 + 9 + 13 + 17.5 + 18) / 5
>>> cov = ((6 - xbar) * (7 - ybar) + (8 - xbar) * (9 - ybar) + (10 -
xbar) * (13 - ybar) +
>>>         (14 - xbar) * (17.5 - ybar) + (18 - xbar) * (18 - ybar)) /
4
>>> print cov
>>> import numpy as np
>>> print np.cov([6, 8, 10, 14, 18], [7, 9, 13, 17.5, 18])[0][1]
22.65
22.65
```

Now that we have calculated the variance of our explanatory variable and the covariance of the response and explanatory variables, we can solve β using the following formula:

$$\beta = \frac{\text{cov}(x, y)}{\text{var}(x)}$$

$$\beta = \frac{22.65}{23.2} = 0.976293103448275 8$$

Having solved β, we can solve α using the following formula:

$$\alpha = \overline{y} - \beta \overline{x}$$

In the preceding formula, $\overline{y}$ is the mean of y and $\overline{x}$ is the mean of x. $(\overline{x}, \overline{y})$ are the coordinates of the centroid, a point that the model must pass through. We can use the centroid and the value of β to solve for α as follows:

$$\alpha = 12.9 - 0.9762931034482758 \times 11.2 = 1.9655172413793114$$

Now that we have solved the values of the model's parameters that minimize the cost function, we can plug in the diameters of the pizzas and predict their prices. For instance, an 11-inch pizza is expected to cost around $12.70, and an 18-inch pizza is expected to cost around $19.54. Congratulations! You used simple linear regression to predict the price of a pizza.

4 Anatomy of a Learning Algorithm

4.1 Building Blocks of a Learning Algorithm

You may have noticed by reading the previous chapter that each learning algorithm we saw consisted of three parts:

1) a loss function;
2) an optimization criterion based on the loss function (a cost function, for example); and
3) an optimization routine that leverages training data to find a solution to the optimization criterion.

These are the building blocks of any learning algorithm. You saw in the previous chapter that some algorithms were designed to explicitly optimize a specific criterion (both linear and logistic regressions, SVM). Some others, including decision tree learning and kNN, optimize the criterion implicitly. Decision tree learning and kNN are among the oldest machine learning algorithms and were invented experimentally based on intuition, without a specific global optimization criterion in mind, and (like it often happens in scientific history) the optimization criteria were developed later to explain why those algorithms work.

By reading modern literature on machine learning, you often encounter references to **gradient descent** or **stochastic gradient descent**. These are two most frequently used optimization algorithms used in cases where the optimization criterion is differentiable.

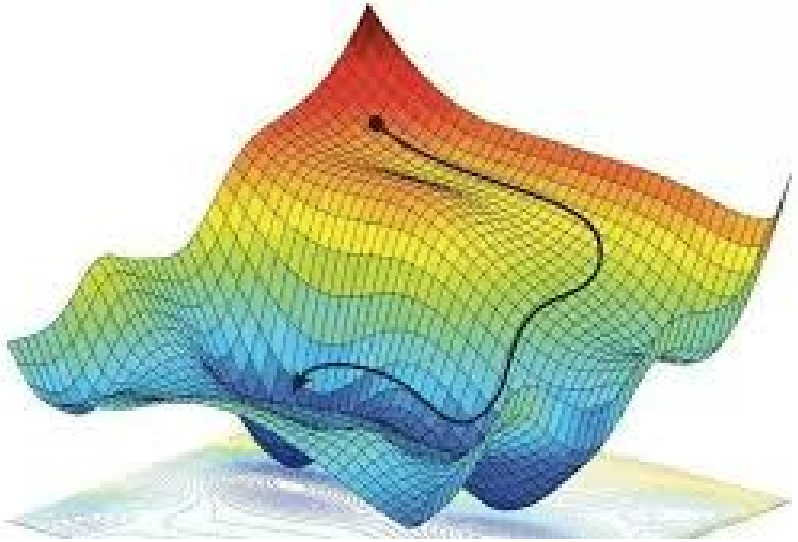
Gradient descent is an iterative optimization algorithm for finding the minimum of a function. To find a *local* minimum of a function using gradient descent, one starts at some random point and takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

Gradient descent can be used to find optimal parameters for linear and logistic regression, SVM and also neural networks which we consider later. For many models, such as logistic regression or SVM, the optimization criterion is *convex*. Convex functions have only one minimum, which is global. Optimization criteria for neural networks are not convex, but in practice even finding a local minimum suffices.

Let's see how gradient descent works.

4.2 Gradient Descent

In this section, I demonstrate how gradient descent finds the solution to a linear regression problem[1]. I illustrate my description with Python source code as well as with plots that show how the solution improves after some iterations of the gradient descent algorithm.



I use a dataset with only one feature. However, the optimization criterion will have two parameters: w and b. The extension to multi-dimensional training data is straightforward: you have variables $w^{(1)}$, $w^{(2)}$, and b for two-dimensional data, $w^{(1)}$, $w^{(2)}$, $w^{(3)}$, and b for three-dimensional data and so on.

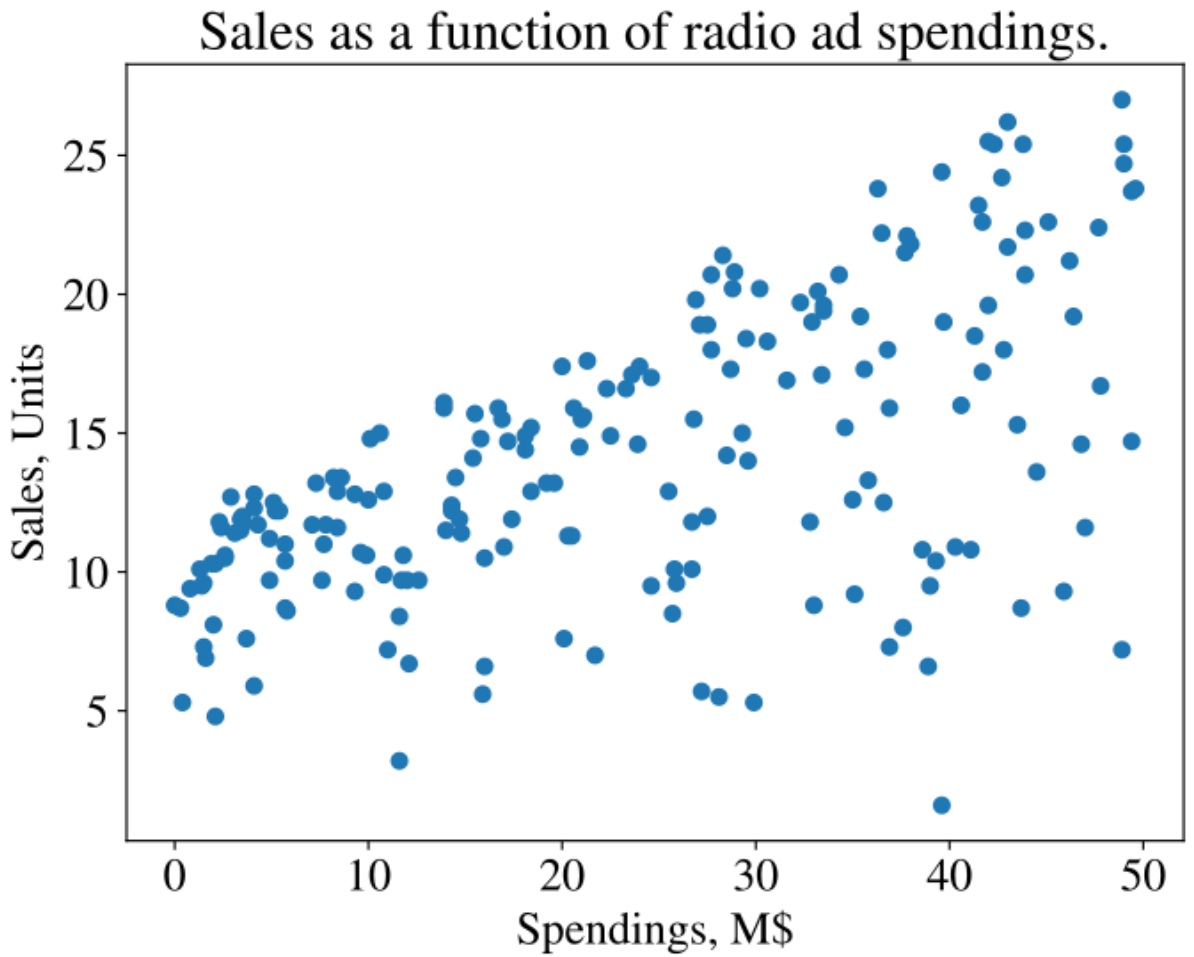


Figure 1: The original data. The Y-axis corresponds to the sales in units (the quantity we want to predict), the X-axis corresponds to our feature: the spendings on radio ads in M$.

To give a practical example, I use the real dataset with the following columns: the Spendings of various companies on radio advertising each year and their annual Sales in terms of units sold. We want to build a regression model that we can use to predict units sold based on how much a company spends on radio advertising. Each row in the dataset represents one specific company:

Company	Spendings, M$	Sales, Units
1	37.8	22.1
2	39.3	10.4
3	45.9	9.3
4	41.3	18.5
..	..	..

We have data for 200 companies, so we have 200 training examples. Fig. 1 shows all examples on a 2D plot.

Remember that the linear regression model looks like this: $f(x) = wx + b$. We don't know what the optimal values for w and b are and we want to learn them from data. To do that, we look for such values for w and b that minimize the mean squared error:

$$l = \frac{1}{N}\sum_{i=1}^{N}(y_i - (wx_i + b))^2.$$

Gradient descent starts with calculating the partial derivative for every parameter:

$$\begin{aligned}\frac{\partial l}{\partial w} &= \frac{1}{N}\sum_{i=1}^{N} -2x_i(y_i - (wx_i + b));\\ \frac{\partial l}{\partial b} &= \frac{1}{N}\sum_{i=1}^{N} -2(y_i - (wx_i + b)).\end{aligned} \tag{1}$$

To find the partial derivative of the term $(y_i - (wx + b))^2$ with respect to w we applied the *chain rule*. Here, we have the chain $f = f_2(f_1)$ where $f_1 = y_i - (wx + b)$ and $f_2 = f_1^2$. To find a partial derivative of f with respect to w we have to first find the partial derivative of f with respect to f_2 which is equal to $2(y_i - (wx + b))$ (from calculus, we know that the derivative $\frac{\partial f}{\partial x}x^2 = 2x$) and then we have to multiply it by the partial derivative of $y_i - (wx + b)$ with respect to w which is equal to $-x$. So overall $\frac{\partial l}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} -2x_i(y_i - (wx_i + b))$. In a similar way, the partial derivative of l with respect to b, $\frac{\partial l}{\partial b}$, was calculated.

We initialize[2] $w_0 = 0$ and $b_0 = 0$ and then iterate through our training examples, each example having the form of $(x_i, y_i) = (Spendings_i, Sales_i)$. For each training example, we update w and b using our partial derivatives. The learning rate α controls the size of an update:

$$\begin{aligned}w_i &\leftarrow \alpha\frac{-2x_i(y_i - (w_{i-1}x_i + b_{i-1}))}{N};\\ b_i &\leftarrow \alpha\frac{-2(y_i - (w_{i-1}x_i + b_{i-1}))}{N},\end{aligned} \tag{2}$$

where w_i and b_i denote the values of w and b after using the example (x_i, y_i) for the update.

One pass through all training examples is called an **epoch**. Typically, we need multiple epochs until we start seeing that the values for w and b don't change much; then we stop.

The function that updates the parameters w and b during one epoch is shown below:

```
def update_w_and_b(spendings, sales, w, b, alpha):
    dl_dw = 0.0
    dl_db = 0.0
    N = len(spendings)

    for i in range(N):
        dl_dw += -2*spendings[i]*(sales[i] - (w*spendings[i] + b))
        dl_db += -2*(sales[i] - (w*spendings[i] + b))

    # update w and b
    w = w - (1/float(N))*dl_dw*alpha
    b = b - (1/float(N))*dl_db*alpha

    return w, b
```

The function that loops over multiple epochs is shown below:

```
def train(spendings, sales, w, b, alpha, epochs):
    for e in range(epochs):
        w, b = update_w_and_b(spendings, sales, w, b, alpha)

        # log the progress
        if e % 400 == 0:
            print("epoch:", e, "loss: ", avg_loss(spendings, sales, w, b))

    return w, b
```

The function *avg_loss* in the above code snippet is a function that computes the mean squared error. It is defined as:

```
def avg_loss(spendings, sales, w, b):
    N = len(spendings)
    total_error = 0.0
    for i in range(N):
        total_error += (sales[i] - (w*spendings[i] + b))**2
    return total_error / float(N)
```

If we run the *train* function for $\alpha = 0.001$, $w = 0.0$, $b = 0.0$, and 15000 epochs, we will see the following output (shown partially):

```
epoch:  0 loss: 92.32078294903626
epoch:  400 loss: 33.79131790081576
```

```
epoch:  800 loss: 27.9918542960729
epoch:  1200 loss: 24.33481690722147
epoch:  1600 loss: 22.028754937538633
...
epoch:  2800 loss: 19.07940244306619
```

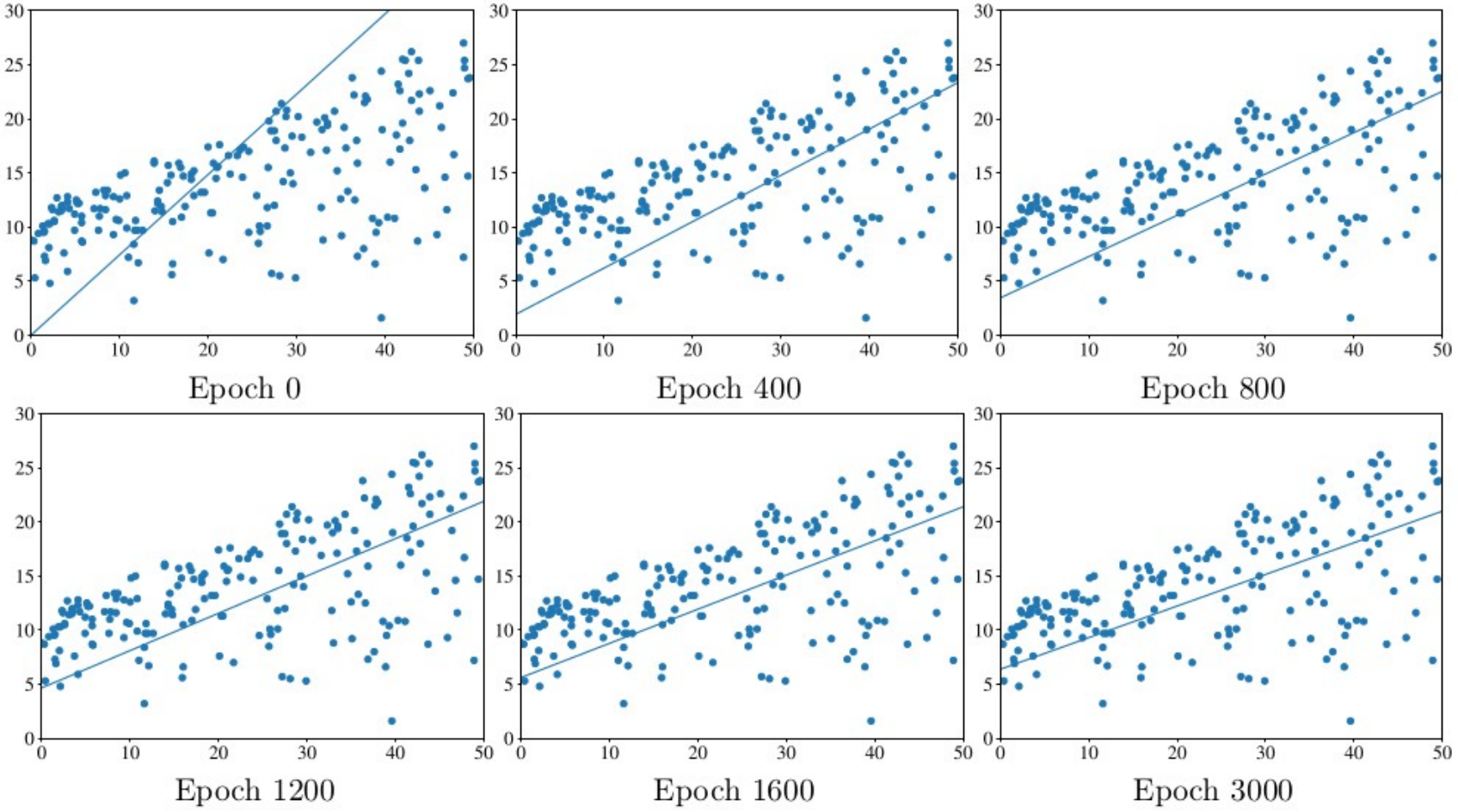


Figure 2: The evolution of the regression line through gradient descent epochs.

You can see that the average loss decreases as the *train* function loops through epochs. Fig. 2 shows the evolution of the regression line through epochs.

Finally, once we have found the optimal values of parameters w and b, the only missing piece is a function that makes predictions:

```
def predict(x, w, b):
    return w*x + b
```

Try to execute the following code:

```
w, b = train(x, y, 0.0, 0.0, 0.001, 15000)
x_new = 23.0
y_new = predict(x_new, w, b)
print(y_new)
```

You can see that the average loss decreases as the *train* function loops through epochs. Fig. 2 shows the evolution of the regression line through epochs.

Finally, once we have found the optimal values of parameters w and b, the only missing piece is a function that makes predictions:

```
def predict(x, w, b):
    return w*x + b
```

Try to execute the following code:

```
w, b = train(x, y, 0.0, 0.0, 0.001, 15000)
x_new = 23.0
y_new = predict(x_new, w, b)
print(y_new)
```

The output is 13.97.

The gradient descent algorithm is sensitive to the choice of the step α. It is also slow for large datasets. Fortunately, several significant improvements to this algorithm have been proposed.

Stochastic gradient descent (SGD) is a version of the algorithm that speeds up the computation by approximating the gradient using smaller batches (subsets) of the training data. SGD itself has various "upgrades". **Adagrad** is a version of SGD that scales α for each parameter according to the history of gradients. As a result, α is reduced for very large gradients and vice-versa. **Momentum** is a method that helps accelerate SGD by orienting the gradient descent in the relevant direction and reducing oscillations. In neural network training, variants of SGD such as **RMSprop** and **Adam**, are most frequently used.

Notice that gradient descent and its variants are not machine learning algorithms. They are solvers of minimization problems in which the function to minimize has a gradient in most points of its domain.

4.3 How Machine Learning Engineers Work

Unless you are a research scientist or work for a huge corporation with a large R&D budget, you usually don't implement machine learning algorithms yourself. You don't implement gradient descent or some other solver either. You use libraries, most of which are open source. A library is a collection of algorithms and supporting tools implemented with stability and efficiency in mind. The most frequently used in practice open-source machine learning library is *scikit-learn*. It's written in Python and C. Here's how you do linear regression in scikit-learn:

```
def train(x, y):
    from sklearn.linear_model import LinearRegression
    model = LinearRegression().fit(x,y)
    return model

model = train(x,y)

x_new = 23.0
y_new = model.predict(x_new)
print(y_new)
```

The output will, again, be 13.97. Easy, right? You can replace LinearRegression with some other type of regression learning algorithm without modifying anything else. It just works. The same can be said about classification. You can easily replace *LogisticRegression* algorithm with *SVC* algorithm (this is scikit-learn's name for the Support Vector Machine algorithm), *DecisionTreeClassifier*, *NearestNeighbors* or many other classification learning algorithms implemented in scikit-learn.

4.4 Learning Algorithms' Particularities

Here we outline some practical particularities that can differentiate one learning algorithm from another. You already know that different learning algorithms can have different hyperparameters (C in SVM, ϵ and d in ID3). Solvers such as gradient descent can also have hyperparameters, like α for example.

Some algorithms, like decision tree learning, can accept categorical features. For example, if you have a feature "color" that can take values "red", "yellow", or "green", you can keep this feature as is. SVM, logistic and linear regression, as well as kNN (with cosine similarity or Euclidean distance metrics), expect numerical values for all features. All algorithms implemented in scikit-learn expect numerical features.

Some algorithms, like SVM, allow the data analyst to provide weightings for each class. These weightings influence how the decision boundary is drawn. If the weight of some class is high, the learning algorithm tries to not make errors in predicting training examples of this class (typically, for the cost of making an error elsewhere). That could be important if instances of some class are in the minority in your training data, but you would like to avoid misclassifying examples of that class as much as possible.

Some classification models, like SVM and kNN, given a feature vector only output the class. Others, like logistic regression or decision trees, can also return the score between 0 and 1 which can be interpreted as either how confident the model is about the prediction or as the probability that the input example belongs to a certain class[3].

Some classification algorithms (like decision tree learning, logistic regression, or SVM) build the model using the whole dataset at once. If you have got additional labeled examples, you have to rebuild the model from scratch. Other algorithms (such as Naïve Bayes, multilayer perceptron, SGDClassifier/SGDRegressor, PassiveAggressiveClassifier/PassiveAggressiveRegressor in scikit-learn) can be trained iteratively, one batch at a time. Once new training examples are available, you can update the model using only the new data.

Finally, some algorithms, like decision tree learning, SVM, and kNN can be used for both classification and regression, while others can only solve one type of problem: either classification or regression, but not both.

Usually, each library provides the documentation that explains what kind of problem each algorithm solves, what input values are allowed and what kind of output the model produces. The documentation also provides information on hyperparameters.