

instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance. On the other hand, $-\log(t)$ is close to 0 when t is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

The cost function over the whole training set is simply the average cost over all training instances. It can be written in a single expression (as you can verify easily), called the *log loss*, shown in [Equation 4-17](#).

Equation 4-17. Logistic Regression cost function (log loss)

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

The bad news is that there is no known closed-form equation to compute the value of θ that minimizes this cost function (there is no equivalent of the Normal Equation). But the good news is that this cost function is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough). The partial derivatives of the cost function with regards to the j^{th} model parameter θ_j is given by [Equation 4-18](#).

Equation 4-18. Logistic cost function partial derivatives

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m \left(\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

This equation looks very much like [Equation 4-5](#): for each instance it computes the prediction error and multiplies it by the j^{th} feature value, and then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives you can use it in the Batch Gradient Descent algorithm. That's it: you now know how to train a Logistic Regression model. For Stochastic GD you would of course just take one instance at a time, and for Mini-batch GD you would use a mini-batch at a time.

Decision Boundaries

Let's use the iris dataset to illustrate Logistic Regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris-Setosa, Iris-Versicolor, and Iris-Virginica (see [Figure 4-22](#)).

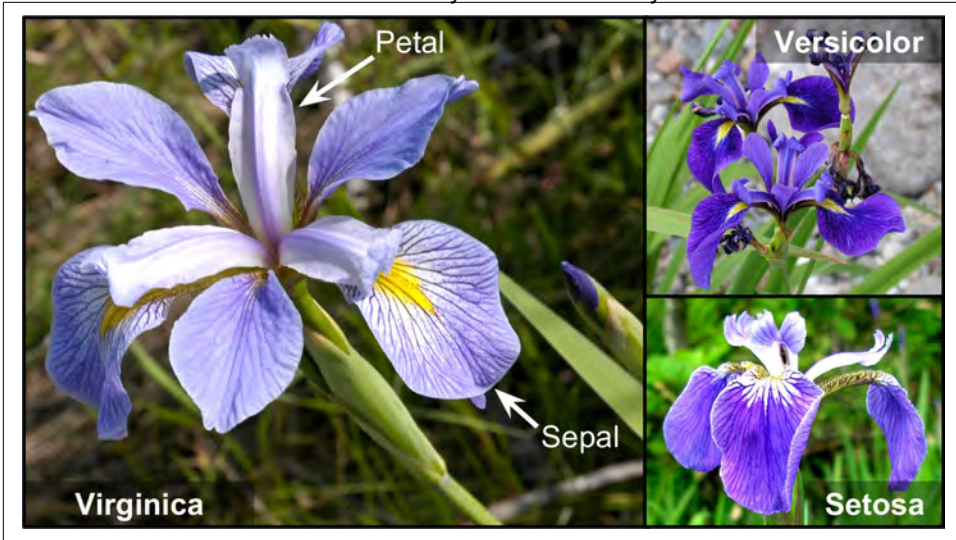


Figure 4-22. Flowers of three iris plant species¹⁶

Let's try to build a classifier to detect the Iris-Virginica type based only on the petal width feature. First let's load the data:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:] # petal width
>>> y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0
```

Now let's train a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 to 3 cm (Figure 4-23):

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
# + more Matplotlib code to make the image look pretty
```

¹⁶ Photos reproduced from the corresponding Wikipedia pages. Iris-Virginica photo by Frank Mayfield (Creative Commons BY-SA 2.0), Iris-Versicolor photo by D. Gordon E. Robertson (Creative Commons BY-SA 3.0), and Iris-Setosa photo is public domain.

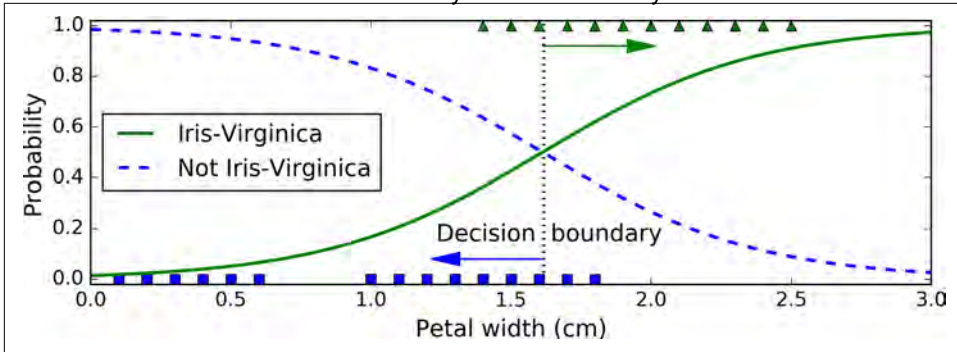


Figure 4-23. Estimated probabilities and decision boundary

The petal width of Iris-Virginica flowers (represented by triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an Iris-Virginica (it outputs a high probability to that class), while below 1 cm it is highly confident that it is not an Iris-Virginica (high probability for the “Not Iris-Virginica” class). In between these extremes, the classifier is unsure. However, if you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely. Therefore, there is a *decision boundary* at around 1.6 cm where both probabilities are equal to 50%: if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an Iris-Virginica, or else it will predict that it is not (even if it is not very confident):

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

Figure 4-24 shows the same dataset but this time displaying two features: petal width and length. Once trained, the Logistic Regression classifier can estimate the probability that a new flower is an Iris-Virginica based on these two features. The dashed line represents the points where the model estimates a 50% probability: this is the model’s decision boundary. Note that it is a linear boundary.¹⁷ Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top-right line have an over 90% chance of being Iris-Virginica according to the model.

¹⁷ It is the set of points \mathbf{x} such that $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, which defines a straight line.

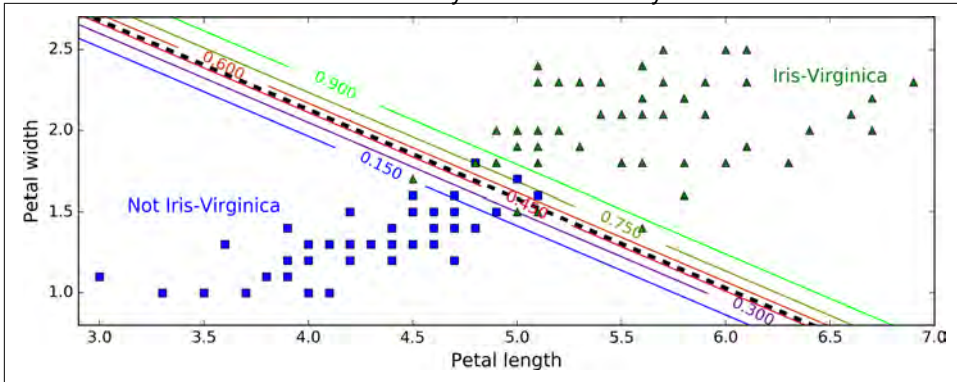


Figure 4-24. Linear decision boundary

Just like the other linear models, Logistic Regression models can be regularized using ℓ_1 or ℓ_2 penalties. Scikit-Learn actually adds an ℓ_2 penalty by default.



The hyperparameter controlling the regularization strength of a Scikit-Learn LogisticRegression model is not alpha (as in other linear models), but its inverse: C. The higher the value of C, the *less* the model is regularized.

Softmax Regression

The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (as discussed in [Chapter 3](#)). This is called *Softmax Regression*, or *Multinomial Logistic Regression*.

The idea is quite simple: when given an instance \mathbf{x} , the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k , then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores. The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for Linear Regression prediction (see [Equation 4-19](#)).

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = \theta_k^T \cdot \mathbf{x}$$

Note that each class has its own dedicated parameter vector θ_k . All these vectors are typically stored as rows in a *parameter matrix* Θ .

Once you have computed the score of every class for the instance \mathbf{x} , you can estimate the probability \hat{p}_k that the instance belongs to class k by running the scores through