Chapter 6. Logistic Regression and Classification

In this chapter we are going to cover *logistic regression*, a type of regression that predicts a probability of an outcome given one or more independent variables. This in turn can be used for *classification*, which is predicting categories rather than real numbers as we did with linear regression.

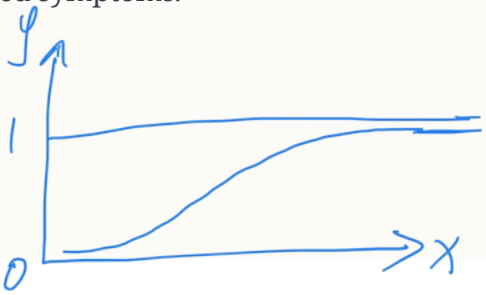
We are not always interested in representing variables as *continuous*, where they can represent an infinite number of real decimal values. There are situations where we would rather variables be *discrete*, or representative of whole numbers, integers, or booleans (1/0, true/false). Logistic regression is trained on an output variable that is discrete (a binary 1 or 0) or a categorical number (which is a whole number). It does output a continuous variable in the form of probability, but that can be converted into a discrete value with a threshold.

Logistic regression is easy to implement and fairly resilient against outliers and other data challenges. Many machine learning problems can best be solved with logistic regression, offering more practicality and performance than other types of supervised machine learning.

Just like we did in Chapter 5 when we covered linear regression, we will attempt to walk the line between statistics and machine learning, using tools and analysis from both disciplines. Logistic regression will integrate many concepts we have learned from this book, from probability to linear regression.

Understanding Logistic Regression

Imagine there was a small industrial accident and you are trying to understand the impact of chemical exposure. You have 11 patients who were exposed for differing numbers of hours to this chemical (please note this is fabricated data). Some have shown symptoms (value of 1) and others have not shown symptoms (value of 0). Let's plot them in Figure 6-1, where the x-axis is hours of exposure and the y-axis is whether or not (1 or 0) they have showed symptoms.



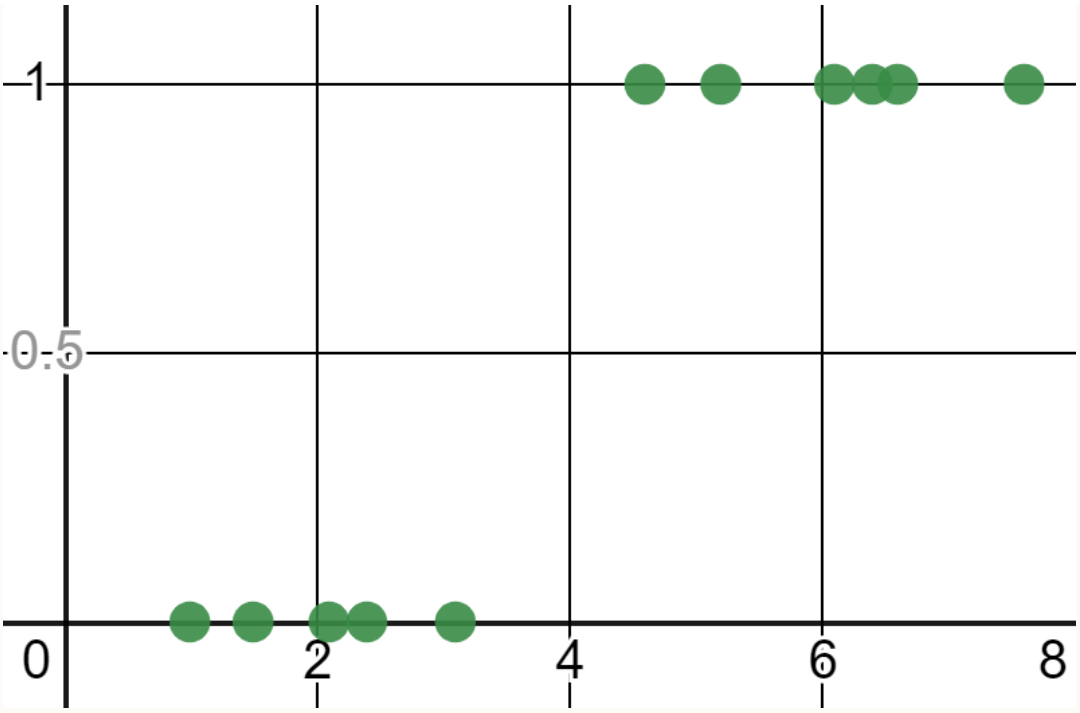


Figure 6-1. Plotting whether patients showed symptoms (1) or not (0) over *x* hours of exposure

At what length of time do patients start showing symptoms? Well it is easy to see at almost four hours, we immediately transition from patients not showing symptoms (0) to showing symptoms (1). In Figure 6-2, we see the same data with a predictive curve.

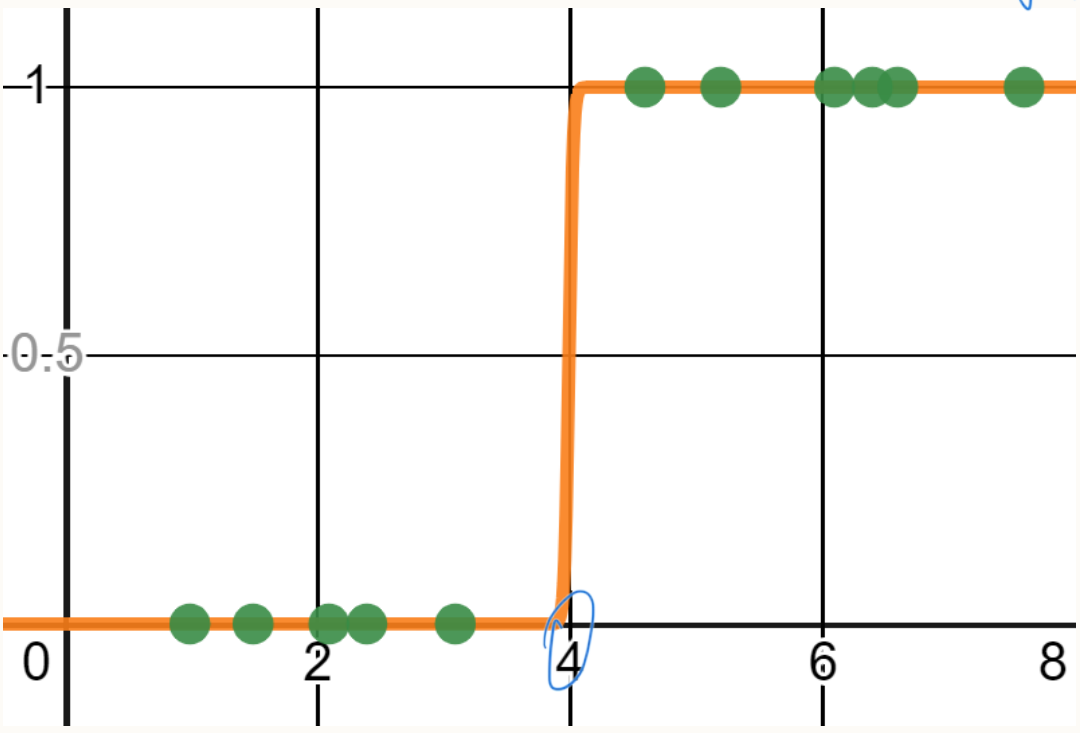


Figure 6-2. After four hours, we see a clear jump where patients start showing symptoms

Doing a cursory analysis on this sample, we can say that there is nearly 0% probability a patient

exposed for fewer than four hours will show symptoms, but there is 100% probability for greater than four hours. Between these two groups, there is an immediate jump to showing symptoms at approximately four hours.

Of course, nothing is ever this clear-cut in the real world. Let's say you gathered more data, where the middle of the range has a mix of patients showing symptoms versus not showing symptoms as shown in Figure 6-3.

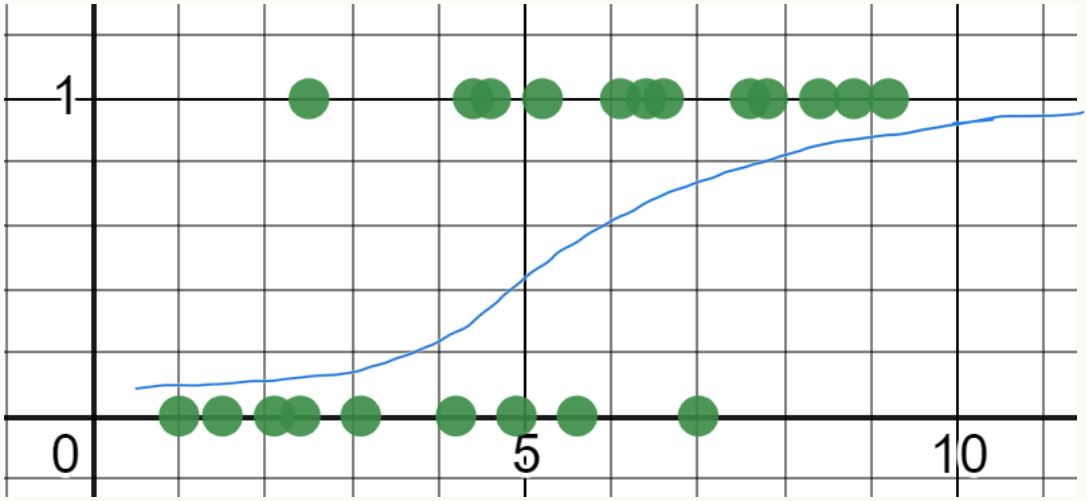


Figure 6-3. A mix of patients who show symptoms (1) and do not show symptoms (0) exists in the middle

The way to interpret this is the probability of patients showing symptoms gradually increases with each hour of exposure. Let's visualize this with a *logistic function*, or an S-shaped curve where the output variable is squeezed between 0 and 1, as shown in Figure 6-4.

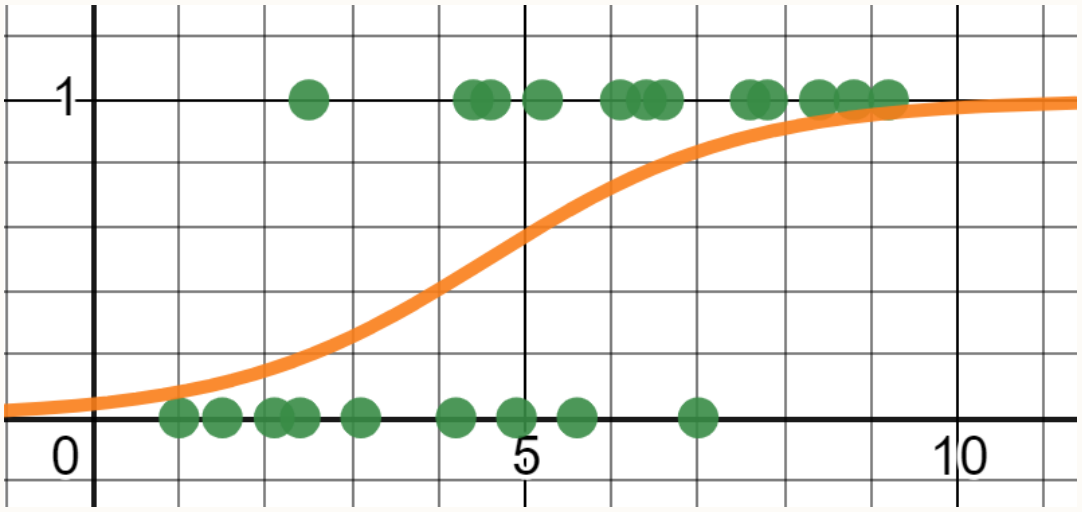


Figure 6-4. Fitting a logistic function to the data

Because of this overlap of points in the middle, there is no distinct cutoff when patients show symptoms but rather a gradual transition from 0% probability to 100% probability (0 and 1). This

example demonstrates how a *logistic regression* results in a curve indicating a probability of belonging to the true category (a patient showed symptoms) across an independent variable (hours of exposure).

We can repurpose a logistic regression to not just predict a probability for given input variables but also add a threshold to predict whether it belongs to that category. For example, if I get a new patient and find they have been exposed for six hours, I predict a 71.1% chance they will show symptoms as traced in Figure 6-5. If my threshold is at least 50% probability to show symptoms, I will simply classify that the patient will show symptoms.

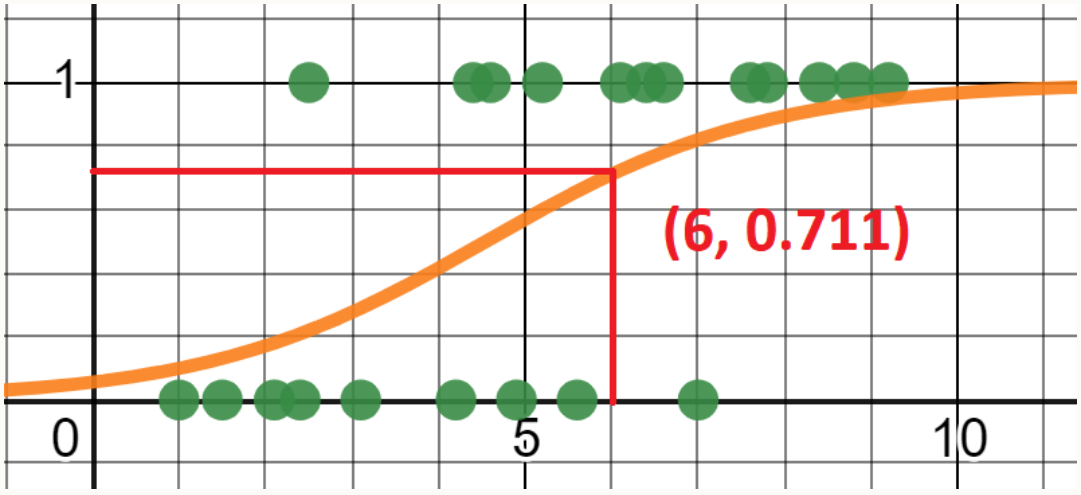


Figure 6-5. We can expect a patient exposed for six hours to be 71.1% likely to have symptoms, and because that's greater than a threshold of 50% we predict they will show symptoms

Performing a Logistic Regression

So how do we perform a logistic regression? Let's first take a look at the logistic function and explore the math behind it.

Logistic Function

The *logistic function* is an S-shaped curve (also known as a *sigmoid curve*) that, for a given set of input variables, produces an output variable between 0 and 1. Because the output variable is between 0 and 1 it can be used to represent a probability.

Here is the logistic function that outputs a probability y for one input variable x:

$$y = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

Note this formula uses Euler's number e, which we covered in Chapter 1. The x variable is the in-

dependent/input variable. β_0 and β_1 are the coefficients we need to solve for.

β_0 and β_1 are packaged inside an exponent resembling a linear function, which you may recall looks identical to $y = mx + b$ or $y = \beta_0 + \beta_1 x$. This is not a coincidence; logistic regression actually has a close relationship to linear regression, which we will discuss later in this chapter. β_0 indeed is the intercept (which we call b in a simple linear regression) and β_1 is the slope for x (which we call m in a simple linear regression). This linear function in the exponent is known as the log-odds function, but for now just know this whole logistic function produces this S-shaped curve we need to output a shifting probability across an x-value.

To declare the logistic function in Python, use the `exp()` function from the `math` package to declare the e exponent as shown in Example 6-1.

Example 6-1. The logistic function in Python for one independent variable

```
import math

def predict_probability(x, b0, b1):
    p = 1.0 / (1.0 + math.exp(-(b0 + b1 * x)))
    return p
```

Let's plot to see what it looks like, and assume B_0 = –2.823 and B_1 = 0.62. We will use SymPy in Example 6-2 and the output graph is shown in Figure 6-6.

Example 6-2. Using SymPy to plot a logistic function

```
from sympy import *
b0, b1, x = symbols('b0 b1 x')

p = 1.0 / (1.0 + exp(-(b0 + b1 * x)))

p = p.subs(b0,-2.823)
p = p.subs(b1, 0.620)
print(p)

plot(p)
```

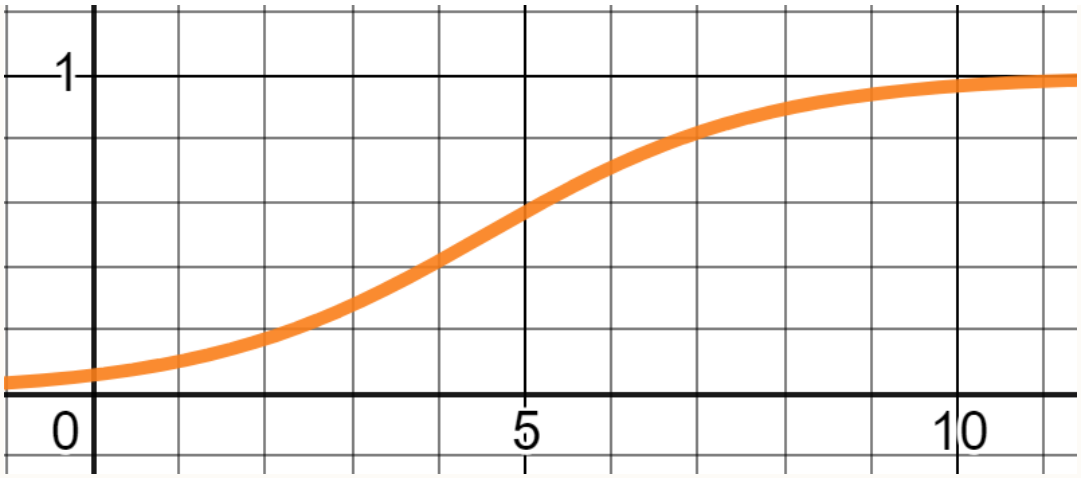


Figure 6-6. A logistic function

In some textbooks, you may alternatively see the logistic function declared like this:

$$p = \frac{e^{\beta_0+\beta_1 x}}{1 + e^{\beta_0+\beta 1 x}}$$

Do not fret about it, because it is the same function, just algebraically expressed differently. Note like linear regression we can also extend logistic regression to more than one input variable ($x_1, x_2, \ldots x_n$), as shown in this formula. We just add more β_x coefficients :

$$p = \frac{1}{1 + e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots \beta_n x_n)}}$$

Fitting the Logistic Curve

How do you fit the logistic curve to a given training dataset? First, the data can have any mix of decimal, integer, and binary variables, but the output variable must be binary (0 or 1). When we actually do prediction, the output variable will be between 0 and 1, resembling a probability .

The data provides our input and output variable values, but we need to solve for the β_0 and β_1 coefficients to fit our logistic function. Recall how we used least squares in Chapter 5. However, this does not apply here. Instead we use *maximum likelihood estimation*, which, as the name suggests, maximizes the likelihood a given logistic curve would output the observed data .

To calculate the maximum likelihood estimation, there really is no closed form equation like in linear regression. We can still use gradient descent, or have a library do it for us. Let's cover both of these approaches starting with the library SciPy.

Using SciPy

The nice thing about SciPy is the models often have a standardized set of functions and APIs,

meaning in many cases you can copy/paste your code and can then reuse it between models. In Example 6-3 you will see a logistic regression performed on our patient data. If you compare it to our linear regression code in Chapter 5, you will see it has nearly identical code in importing, separating, and fitting our data. The main difference is I use a `LogisticRegression()` for my model instead of a `LinearRegression()`.

Example 6-3. Using a plain logistic regression in SciPy

```
import pandas as pd
from sklearn.linear_model import LogisticRegression

# Load the data
df = pd.read_csv('https://bit.ly/33ebs2R', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Perform logistic regression
# Turn off penalty
model = LogisticRegression(penalty='none')
model.fit(X, Y)

# print beta1
print(model.coef_.flatten()) # 0.69267212

# print beta0
print(model.intercept_.flatten()) # -3.17576395
```

MAKING PREDICTIONS

To make specific predictions, use the `predict()` and `predict_prob()` functions on the `model` object in SciPy, whether it is a `LogisticRegression` or any other type of classification model. The `predict()` function will predict a specific class (e.g., True 1.0 or False 1.0) while the `predict_prob()` will output probabilities for each class.

After running the model in SciPy, I get a logistic regression where β_0 = –3.17576395 and β_1 = 0.69267212. When I plot this, it should look pretty good as shown in Figure 6-7.

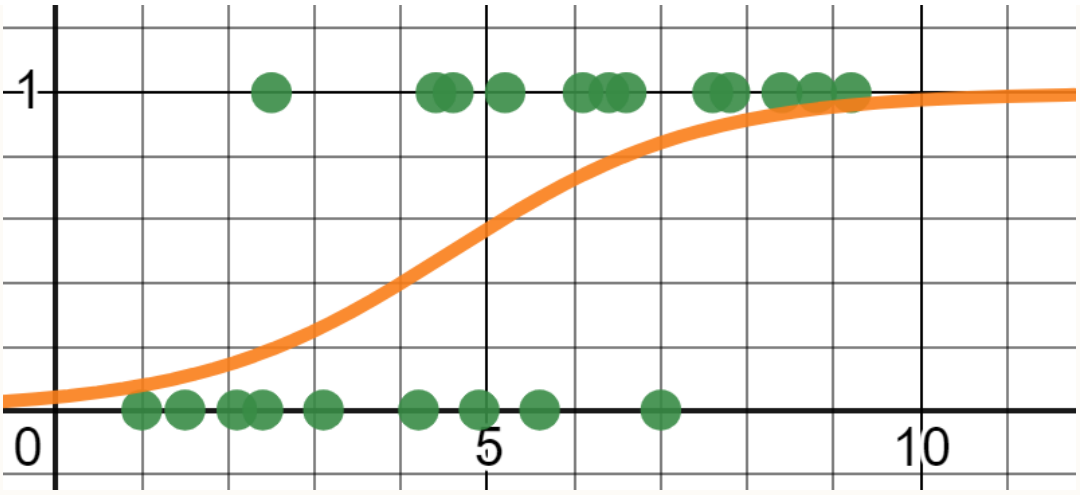


Figure 6-7. Plotting the logistic regression

There are a couple of things to note here. When I created the `LogisticRegression()` model, I specified no `penalty` argument, which chooses a regularization technique like `l1` or `l2`. While this is beyond the scope of this book, I have included brief insights in the following note "Learning About SciPy Parameters" so that you have helpful references on hand.

Finally, I am going to `flatten()` the coefficient and intercept, which come out as multidimensional matrices but with one element. *Flattening* means collapsing a matrix of numbers into lesser dimensions, particularly when there are fewer elements than there are dimensions. For example, I use `flatten()` here to take a single number nested into a two-dimensional matrix and pull it out as a single value. I then have my β_0 and β_1 coefficients.

LEARNING ABOUT SCIPY PARAMETERS

SciPy offers a lot of options in its regression and classification models. Unfortunately, there is not enough bandwidth or pages to cover them as this is not a book focusing exclusively on machine learning.

However, the SciPy docs are well-written and the page on logistic regression is found here.

If a lot of terms are unfamiliar, such as regularization and `l1` and `l2` penalties, there are other great O'Reilly books exploring these topics. One of the more helpful texts I have found is *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* by Aurélien Géron.

Using Maximum Likelihood and Gradient Descent

As I have done throughout this book, I aim to provide insights on building techniques from scratch even if libraries can do it for us. There are several ways to fit a logistic regression ourselves, but all methods typically turn to maximum likelihood estimation (MLE). MLE maximizes the likelihood a given logistic curve would output the observed data. It is different than sum of

squares, but we can still apply gradient descent or stochastic gradient descent to solve it.

I'll try to streamline the mathematical jargon and minimize the linear algebra here. Essentially, the idea is to find the β_0 and β_1 coefficients that bring our logistic curve to those points as closely as possible, indicating it is most likely to have produced those points. If you recall from Chapter 2 when we studied probability, we combine probabilities (or likelihoods) of multiple events by multiplying them together. In this application, we are calculating the likelihood we would see all these points for a given logistic regression curve.

Applying the idea of joint probabilities, each patient has a likelihood they would show symptoms *based on the fitted logistic function* as shown in Figure 6-8.

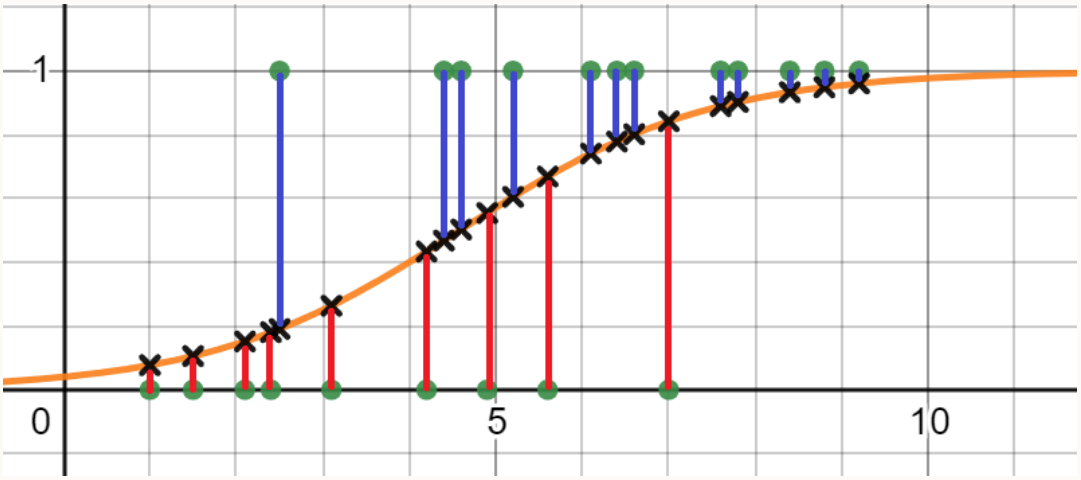


Figure 6-8. Every input value has a corresponding likelihood on the logistic curve

We fetch each likelihood off the logistic regression curve above or below each point. If the point is below the logistic regression curve, we need to subtract the resulting probability from 1.0 because we want to maximize the false cases too.

Given coefficients β_0 = –3.17576395 and β_1 = 0.69267212, Example 6-4 shows how we calculate the joint likelihood for this data in Python.

Example 6-4. Calculating the joint likelihood of observing all the points for a given logistic regression

```
import math
import pandas as pd

patient_data = pd.read_csv('https://bit.ly/33ebs2R', delimiter=",").itertuples()

b0 = -3.17576395
```

```
b1 = 0.69267212

def logistic_function(x):
    p = 1.0 / (1.0 + math.exp(-(b0 + b1 * x)))
    return p

# Calculate the joint likelihood
joint_likelihood = 1.0

for p in patient_data:
    if p.y == 1.0:
        joint_likelihood *= logistic_function(p.x)
    elif p.y == 0.0:
        joint_likelihood *= (1.0 - logistic_function(p.x))

print(joint_likelihood) # 4.7911180221699105e-05
```

Here's a mathematical trick we can do to compress that `if` expression. As we covered in Chapter 1, when you set any number to the power of 0 it will always be 1. Take a look at this formula and note the handling of true (1) and false (0) cases in the exponents:

$$\text{joint likelihood} = \prod_{i=1}^{n} \left(\frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x_i)}} \right)^{y_i} \times \left(\frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x_i)}} \right)^{1.0 - y_i}$$

To do this in Python, compress everything inside that `for` loop into Example 6-5.

Example 6-5. Compressing the joint likelihood calculation without an `if` expression

```
for p in patient_data:
    joint_likelihood *= logistic_function(p.x) ** p.y * \
                        (1.0 - logistic_function(p.x)) ** (1.0 - p.y)
```

What exactly did I do? Notice that there are two halves to this expression, one for when $y = 1$ and the other where $y = 0$. When any number is raised to exponent 0, it will result in 1. Therefore, whether *y* is 1 or 0, it will cause the opposite condition on the other side to evaluate to 1 and have no effect in multiplication. We get to express our `if` expression but do it completely in a mathematical expression. We cannot do derivatives on expressions that use `if`, so this will be helpful.

Note that computers can get overwhelmed multiplying several small decimals together, known as *floating point underflow*. This means that as decimals get smaller and smaller, which can happen

in multiplication, the computer runs into limitations keeping track of that many decimal places. There is a clever mathematical hack to get around this. You can take the `log()` of each decimal you are multiplying and instead add them together. This is thanks to the additive properties of logarithms we covered in Chapter 1. This is more numerically stable, and you can then call the `exp()` function to convert the total sum back to get the product.

Let's revise our code to use logarithmic addition instead of multiplication (see Example 6-6). Note that the `log()` function will default to base e and while any base technically works, this is preferable because e^x is the derivative of itself and will computationally be more efficient.

Example 6-6. Using logarithmic addition

```
# Calculate the joint likelihood
joint_likelihood = 0.0

for p in patient_data:
    joint_likelihood += math.log(logistic_function(p.x) ** p.y * \
                                 (1.0 - logistic_function(p.x)) ** (1.0 - p.y))

joint_likelihood = math.exp(joint_likelihood)
```

To express the preceding Python code in mathematical notation:

$$\text{joint likelihood} = \sum_{i=1}^{n} log\left(\left(\frac{1.0}{1.0 + e^{-(\beta_0+\beta_1 x_i)}}\right)^{y_i} \times \left(1.0 - \frac{1.0}{1.0 + e^{-(\beta_0+\beta_1 x_i)}}\right)^{1.0-y_i}\right)$$

Would you like to calculate the partial derivatives for β_0 and β_1 in the preceding expression? I didn't think so. It's a beast. Goodness, expressing that function in SymPy alone is a mouthful! Look at this in Example 6-7.

Example 6-7. Expressing a joint likelihood for logistic regression in SymPy

```
joint_likelihood = Sum(log((1.0 / (1.0 + exp(-(b + m * x(i)))))**y(i) * \
        (1.0 - (1.0 / (1.0 + exp(-(b + m * x(i))))))**(1-y(i))), (i, 0, n))
```

So let's just allow SymPy to do the partial derivatives for us, for β_0 and β_1 respectively. We will then immediately compile and use them for gradient descent, as shown in Example 6-8.

Example 6-8. Using gradient descent on logistic regression

```
from sympy import *
```

```
import pandas as pd

points = list(pd.read_csv("https://tinyurl.com/y2cocoo7").itertuples())

b1, b0, i, n = symbols('b1 b0 i n')
x, y = symbols('x y', cls=Function)
joint_likelihood = Sum(log((1.0 / (1.0 + exp(-(b0 + b1 * x(i))))) ** y(i) \
        * (1.0 - (1.0 / (1.0 + exp(-(b0 + b1 * x(i)))))) ** (1 - y(i))), (i, 0, n))

# Partial derivative for m, with points substituted
d_b1 = diff(joint_likelihood, b1) \
                  .subs(n, len(points) - 1).doit() \
                  .replace(x, lambda i: points[i].x) \
                  .replace(y, lambda i: points[i].y)

# Partial derivative for m, with points substituted
d_b0 = diff(joint_likelihood, b0) \
                  .subs(n, len(points) - 1).doit() \
                  .replace(x, lambda i: points[i].x) \
                  .replace(y, lambda i: points[i].y)

# compile using lambdify for faster computation
d_b1 = lambdify([b1, b0], d_b1)
d_b0 = lambdify([b1, b0], d_b0)

# Perform Gradient Descent
b1 = 0.01
b0 = 0.01
L = .01

for j in range(10_000):
    b1 += d_b1(b1, b0) * L
    b0 += d_b0(b1, b0) * L

print(b1, b0)
# 0.6926693075370812 -3.175751550409821
```

After calculating the partial derivatives for β_0 and β_1, we substitute the x- and y-values as well as the number of data points *n*. Then we use `lambdify()` to compile the derivative function for efficiency (it uses NumPy behind the scenes). After that, we perform gradient descent like we did in Chapter 5, but since we are trying to maximize rather than minimize, we add each adjustment to β_0 and β_1 rather than subtract like in least squares.

As you can see in Example 6-8, we got β_0 = –3.17575 and β_1 = 0.692667. This is highly comparable to the coefficient values we got in SciPy earlier.

As we learned to do in Chapter 5, we can also use stochastic gradient descent and only sample one or a handful of records on each iteration. This would extend the benefits of increasing computational speed and performance as well as prevent overfitting. It would be redundant to cover it again here, so we will keep moving on.

Multivariable Logistic Regression

Let's try an example that uses logistic regression on multiple input variables. Table 6-1 shows a sample of a few records from a fictitious dataset containing some employment-retention data (full dataset is here).

Table 6-1. Sample of employment-retention data

SEX	AGE	PROMOTIONS	YEARS_EMPLOYED	DID_QUIT
1	32	3	7	0
1	34	2	5	0
1	29	2	5	1
0	42	4	10	0
1	43	4	10	0

There are 54 records in this dataset. Let's say we want to use it to predict whether other employees are going to quit and logistic regression can be utilized here (although none of this is a good idea, and I will elaborate why later). Recall we can support more than one input variable as shown in this formula:

$$y = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_n x_n)}}$$

I will create β coefficients for each of the variables `sex`, `age`, `promotions`, and `years_employed`. The output variable `did_quit` is binary, and that is going to drive the logistic regression outcome we are predicting. Because we are dealing with multiple dimensions, it is go-

ing to be hard to visualize the curvy hyperplane that is our logistic curve. So we will steer clear from visualization.

Let's make it interesting. We will use scikit-learn but make an interactive shell we can test employees with. Example 6-9 shows the code, and when we run it, a logistic regression will be performed, and then we can type in new employees to predict whether they quit or not. What can go wrong? Nothing, I'm sure. We are only making predictions on people's personal attributes and making decisions accordingly. I'm sure it will be fine.

(If it was not clear, I'm being very tongue in cheek).

Example 6-9. Doing a multivariable logistic regression on employee data

```
import pandas as pd
from sklearn.linear_model import LogisticRegression

employee_data = pd.read_csv("https://tinyurl.com/y6r7qjrp")

# grab independent variable columns
inputs = employee_data.iloc[:, :-1]

# grab dependent "did_quit" variable column
output = employee_data.iloc[:, -1]

# build logistic regression
fit = LogisticRegression(penalty='none').fit(inputs, output)

# Print coefficients:
print("COEFFICIENTS: {0}".format(fit.coef_.flatten()))
print("INTERCEPT: {0}".format(fit.intercept_.flatten()))

# Interact and test with new employee data
def predict_employee_will_stay(sex, age, promotions, years_employed):
    prediction = fit.predict([[sex, age, promotions, years_employed]])
    probabilities = fit.predict_proba([[sex, age, promotions, years_employed]])
    if prediction == [[1]]:
        return "WILL LEAVE: {0}".format(probabilities)
    else:
        return "WILL STAY: {0}".format(probabilities)

# Test a prediction
while True:
    n = input("Predict employee will stay or leave {sex},
```

```
        {age},{promotions},{years employed}: ")
    (sex, age, promotions, years_employed) = n.split(",")
    print(predict_employee_will_stay(int(sex), int(age), int(promotions),
          int(years_employed)))
```

Figure 6-9 shows the result whether an employee is predicted to quit. The employee is a sex "1," age is 34, had 1 promotion, and has been at the company for 5 years. Sure enough, the prediction is "WILL LEAVE."

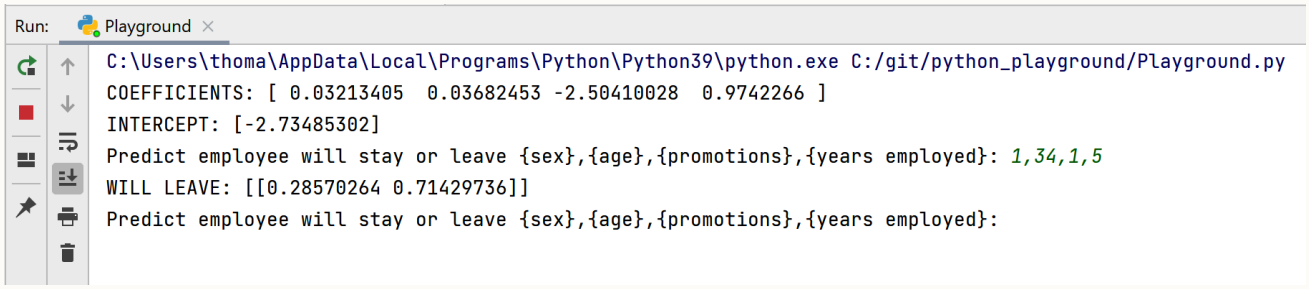


Figure 6-9. Making a prediction whether a 34-year-old employee with 1 promotion and 5 years, employment will quit

Note that the `predict_proba()` function will output two values, the first being the probability of 0 (false) and the second being 1 (true).

You will notice that the coefficients for `sex`, `age`, `promotions`, and `years_employed` are displayed in that order. By the weight of the coefficients, you can see that `sex` and `age` play very little role in the prediction (they both have a weight near 0). However, `promotions` and `years_employed` have significant weights of `-2.504` and `0.97`. Here's a secret with this toy dataset: I fabricated it so that an employee quits if they do not get a promotion roughly every two years. Sure enough, my logistic regression picked up this pattern and you can try it out with other employees as well. However, if you venture outside the ranges of data it was trained on, the predictions will likely start falling apart (e.g., if put in a 70-year-old employee who hasn't been promoted in three years, it's hard to say what this model will do since it has no data around that age).

Of course, real life is not always this clean. An employee who has been at a company for eight years and has never gotten a promotion is likely comfortable with their role and not leaving anytime soon. If that is the case, variables like age then might play a role and get weighted. Then of course we can get concerned about other relevant variables that are not being captured. See the following warning to learn more.

BE CAREFUL MAKING CLASSIFICATIONS ON PEOPLE!

A quick and surefire way to shoot yourself in the foot is to collect data on people and use it to make predictions haphazardly. Not only can data privacy concerns come about, but legal and PR issues can emerge if the model is found to be discriminatory. Input variables like race and gender can become weighted from machine learning training. After that, undesirable outcomes are inflicted on those demographics like not being hired or being denied loans. More extreme applications include being falsely flagged by surveillance systems or being denied criminal parole. Note too that seemingly benign variables like commute time have been found to correlate with discriminatory variables.

At the time of writing, a number of articles have been citing machine learning discrimination as an issue:

- Katyanna Quach, "Teen turned away from roller rink after AI wrongly identifies her as banned troublemaker", *The Register*, July 16, 2021.
- Kashmir Hill, "Wrongfully Accused by an Algorithm", *New York Times*, June 24, 2020.

As data privacy laws continue to evolve, it is advisable to err on the side of caution and engineer personal data carefully. Think about what automated decisions will be propagated and how that can cause harm. Sometimes it is better to just leave a "problem" alone and keep doing it manually.

Finally, on this employee-retention example, think about where this data came from. Yes, I made up this dataset but in the real world you always want to question what process created the data. Over what period of time did this sample come from? How far back do we go looking for employees who quit? What constitutes an employee who stayed? Are they current employees at this point in time? How do we know they are not about to quit, making them a false negative? Data scientists easily fall into traps analyzing only what data says, but not questioning where it came from and what assumptions are built into it.

The best way to get answers to these questions is to understand what the predictions are being used for. Is it to decide when to give people promotions to retain them? Can this create a circular bias promoting people with a set of attributes? Will that bias be reaffirmed when those promotions start becoming the new training data?

These are all important questions, and perhaps even inconvenient ones that cause unwanted scope to creep into the project. If this scrutiny is not welcomed by your team or leadership on a project, consider empowering yourself with a different role where curiosity becomes a strength.

Understanding the Log-Odds

At this point, it is time to discuss the logistic regression and what it is mathematically made of.

This can be a bit dizzying so take your time here. If you get overwhelmed, you can always revisit this section later.

Starting in the 1900s, it has always been of interest to mathematicians to take a linear function and scale its output to fall between 0 and 1, and therefore be useful for predicting probability. The log-odds, also called the logit function, lends itself to logistic regression for this purpose.

Remember earlier I pointed out the exponent value $\beta_0 + \beta_1 x$ is a linear function? Look at our logistic function again:

$$p = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

This linear function being raised to e is known as the *log-odds* function, which takes the logarithm of the odds for the event of interest. Your response might be, "Wait, I don't see any `log()` or odds. I just see a linear function!" Bear with me, I will show the hidden math.

As an example, let's use our logistic regression from earlier where B_0 = -3.17576395 and B_1 = 0.69267212. What is the probability of showing symptoms after six hours, where $x = 6$? We already know how to do this: plug these values into our logistic function:

$$p = \frac{1.0}{1.0 + e^{-(-3.17576395 + 0.69267212(6))}} = 0.727161542928554$$

We plug in these values and output a probability of 0.72716. But let's look at this from an odds perspective. Recall in Chapter 2 we learned how to calculate odds from a probability:

$$\text{odds} = \frac{p}{1 - p}$$

$$\text{odds} = \frac{.72716}{1 - .72716} = 2.66517246407876$$

So at six hours, a patient is 2.66517 times more likely to show symptoms than not show symptoms.

When we wrap the odds function in a natural logarithm (a logarithm with base e), we call this the *logit function*. The output of this formula is what we call the *log-odds*, named...shockingly... because we take the logarithm of the odds:

$$\text{logit} = \log\left(\frac{p}{1 - p}\right)$$

$$\text{logit} = \log\left(\frac{.72716}{1 - .72716}\right) = 0.98026877$$

Our log-odds at six hours is 0.9802687. What does this mean and why do we care? When we are in

"log-odds land" it is easier to compare one set of odds against another. We treat anything greater than 0 as favoring odds an event will happen, whereas anything less than 0 is against an event. A log-odds of –1.05 is linearly the same distance from 0 as 1.05. In plain odds, though, the equivalents are 0.3499 and 2.857, respectively, which is not as interpretable. That is the convenience of log-odds.

ODDS AND LOGS

Logarithms and odds have an interesting relationship. Odds are against an event when it is between 0.0 and 1.0, but anything greater than 1.0 favors the event and extends into positive infinity. This lack of symmetry is awkward. However, logarithms rescale an odds so that it is completely linear, where a log-odds of 0.0 means fair odds. A log-odds of –1.05 is linearly the same distance from 0 as 1.05, thus make comparing odds much easier .

Josh Starmer has a great video talking about this relationship between odds and logs.

Recall I said the linear function in our logistic regression formula $\beta_0 + \beta_1 x$ is our log-odds function. Check this out:

$$\text{log-odds} = \beta_0 + \beta_1 x$$

$$\text{log-odds} = -3.17576395 + 0.69267212(6)$$

$$\text{log-odds} = 0.98026877$$

It's the same value 0.98026877 as our previous calculation, the odds of our logistic regression at x = 6 and then taking the `log()` of it! So what is the link? What ties all this together? Given a probability from a logistic regression p and input variable x, it is this:

$$log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x$$

Let's plot the log-odds line alongside the logistic regression, as shown in Figure 6-10.

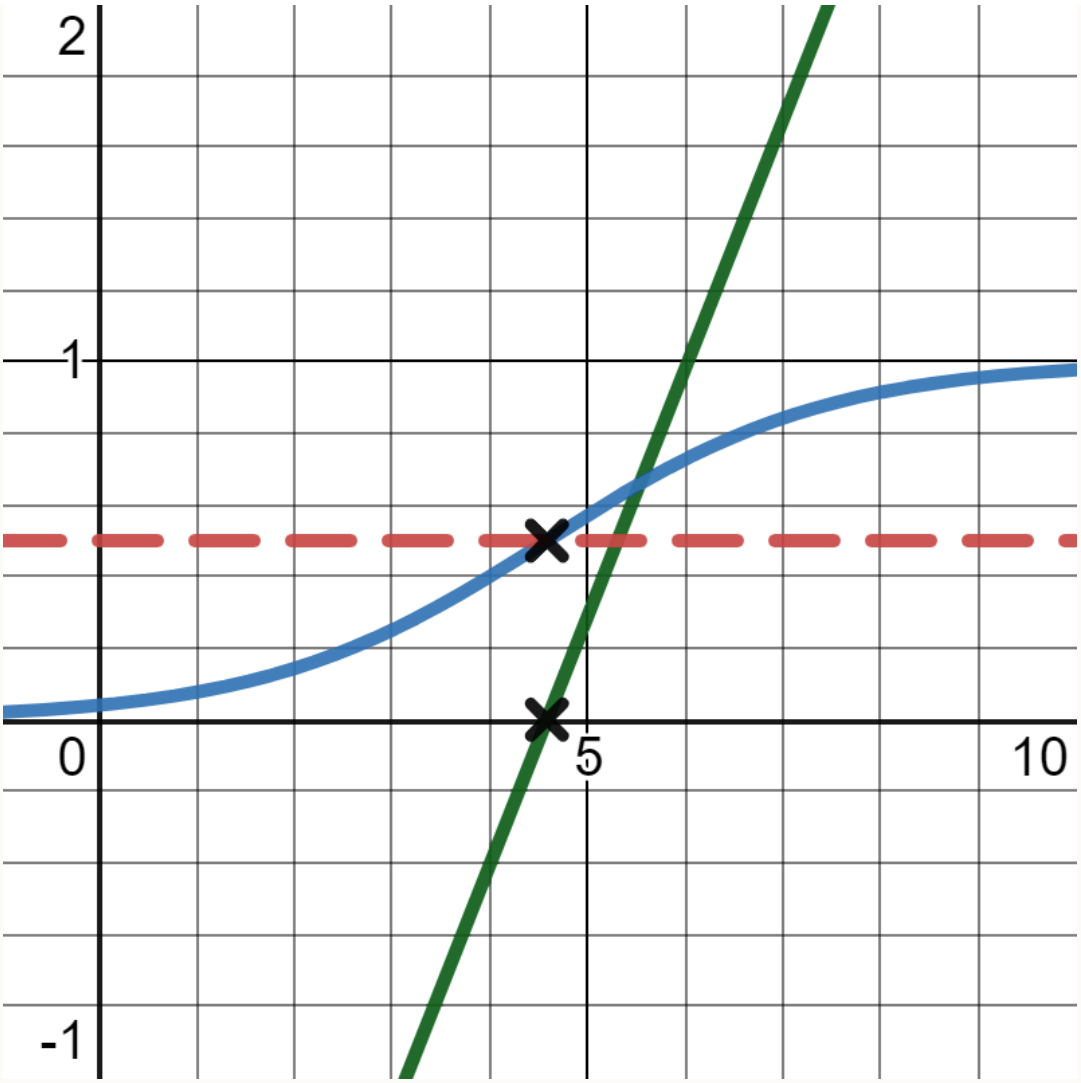


Figure 6-10. The log-odds line is converted into a logistic function that outputs a probability

Every logistic regression is actually backed by a linear function, and that linear function is a log-odds function. Note in Figure 6-10 that when the log-odds is 0.0 on the line, then the probability of the logistic curve is at 0.5. This makes sense because when our odds are fair at 1.0, the probability is going to be 0.50 as shown in the logistic regression, and the log-odds are going to be 0 as shown by the line.

Another benefit we get looking at the logistic regression from an odds perspective is we can compare the effect between one x-value and another. Let's say I want to understand how much my odds change between six hours and eight hours of exposure to the chemical. I can take the odds at six hours and then eight hours, and then ratio the two odds against each other in an *odds ratio*. This is not to be confused with a plain odds which, yes, is a ratio, but it is not an odds ratio.

Let's first find the probabilities of symptoms for six hours and eight hours, respectively:

$$p = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

$$p_6 = \frac{1.0}{1.0 + e^{-(-3.17576395 + 0.69267212(6))}} = 0.727161542928554$$

$$p_8 = \frac{1.0}{1.0 + e^{-(-3.17576395 + 0.69267212(8))}} = 0.914167258137741$$

Now let's convert those into odds, which we will declare as o_x:

$$o = \frac{p}{1 - p}$$

$$o_6 = \frac{0.727161542928554}{1 - 0.727161542928554} = 2.66517246407876$$

$$o_8 = \frac{0.914167258137741}{1 - 0.914167258137741} = 10.6505657200694$$

Finally, set the two odds against each other as an odds ratio, where the odds for eight hours is the numerator and the odds for six hours is in the denominator. We get a value of approximately 3.996, meaning that our odds of showing symptoms increases by nearly a factor of four with an extra two hours of exposure:

$$\text{odds ratio} = \frac{10.6505657200694}{2.66517246407876} = 3.99620132040906$$

You will find this odds ratio value of 3.996 holds across any two-hour range, like 2 hours to 4 hours, 4 hours to 6 hours, 8 hours to 10 hours, and so forth. As long as it's a two-hour gap, you will find that odds ratio stays consistent. It will differ for other range lengths .

R-Squared

We covered quite a few statistical metrics for linear regression in Chapter 5, and we will try to do the same for logistic regression. We still worry about many of the same problems as in linear regression, including overfitting and variance. As a matter of fact, we can borrow and adapt several metrics from linear regression and apply them to logistic regression. Let's start with R^2.

Just like linear regression, there is an R^2 for a given logistic regression. If you recall from Chapter 5, the R^2 indicates how well a given independent variable explains a dependent variable. Applying this to our chemical exposure problem, it makes sense we want to measure how much chemical exposure hours explains showing symptoms.

There is not really a consensus on the best way to calculate the R^2 on a logistic regression, but a popular technique known as McFadden's Pseudo R^2 closely mimics the R^2 used in linear regres-

sion . We will use this technique in the following examples and here is the formula:

$$R^2 = \frac{(\text{log likelihood}) - (\text{log likelihood fit})}{(\text{log likelihood})}$$

We will learn how to calculate the "log likelihood fit" and "log likelihood" so we can calculate the R^2.

We cannot use residuals here like in linear regression, but we can project the outcomes back onto the logistic curve as shown in Figure 6-11, and look up their corresponding likelihoods between 0.0 and 1.0.

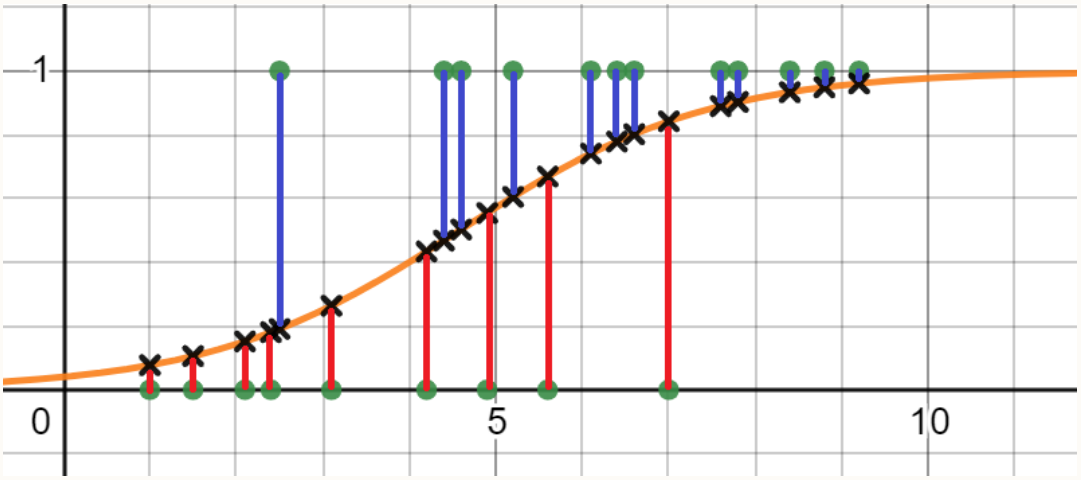


Figure 6-11. Projecting the output values back onto the logistic curve

We can then take the `log()` of each of those likelihoods and sum them together. This will be the log likelihood of the fit (Example 6-10). Just like we did calculating maximum likelihood, we will convert the "false" likelihoods by subtracting from 1.0.

Example 6-10. Calculating the log likelihood of the fit

```
from math import log, exp
import pandas as pd

patient_data = pd.read_csv('https://bit.ly/33ebs2R', delimiter=",").itertuples()

b0 = -3.17576395
b1 = 0.69267212

def logistic_function(x):
    p = 1.0 / (1.0 + exp(-(b0 + b1 * x)))
    return p
```

```
# Sum the log-likelihoods
log_likelihood_fit = 0.0

for p in patient_data:
    if p.y == 1.0:
        log_likelihood_fit += log(logistic_function(p.x))
    elif p.y == 0.0:
        log_likelihood_fit += log(1.0 - logistic_function(p.x))

print(log_likelihood_fit) # -9.946161673231583
```

Using some clever binary multiplication and Python comprehensions, we can consolidate that `for` loop and `if` expression into one line that returns the `log_likelihood_fit`. Similar to what we did in the maximum likelihood formula, we can use some binary subtraction between the true and false cases to mathematically eliminate one or the other. In this case, we multiply by 0 and therefore apply either the true or the false case, but not both, to the sum accordingly (Example 6-11).

Example 6-11. Consolidating our log likelihood logic into a single line

```
log_likelihood_fit = sum(log(logistic_function(p.x)) * p.y +
                         log(1.0 - logistic_function(p.x)) * (1.0 - p.y)
                         for p in patient_data)
```

If we were to express the likelihood of the fit in mathematic notation, this is what it would look like. Note that $f(x_i)$ is the logistic function for a given input variable x_i:

$$\text{log likelihood fit} = \sum_{i=1}^{n} (log\left(f\left(x_i\right)\right) \times y_i) + (log\left(1.0 - f\left(x_i\right)\right) \times (1 - y_i))$$

As calculated in Examples 6-10 and 6-11, we have -9.9461 as our log likelihood of the fit. We need one more datapoint to calculate the R^2: the log likelihood that estimates without using any input variables and simply uses the number of true cases divided by all cases (effectively leaving only the intercept). Note we can count the number of symptomatic cases by summing all the y-values together $\sum y_i$, because only the 1s and not the 0s will count into the sum. Here is the formula:

$$\text{log likelihood} = \frac{\sum y_i}{n} \times y_i + \left(1 - \frac{\sum y_i}{n}\right) \times (1 - y_i)$$

Here is the expanded Python equivalent of this formula applied in Example 6-12.

Example 6-12. Log likelihood of patients

```
import pandas as pd
from math import log, exp

patient_data = list(pd.read_csv('https://bit.ly/33ebs2R', delimiter=",") \
     .itertuples())

likelihood = sum(p.y for p in patient_data) / len(patient_data)

log_likelihood = 0.0

for p in patient_data:
    if p.y == 1.0:
        log_likelihood += log(likelihood)
    elif p.y == 0.0:
        log_likelihood += log(1.0 - likelihood)

print(log_likelihood) # -14.341070198709906
```

To consolidate this logic and reflect the formula, we can compress that `for` loop and `if` expression into a single line, using some binary multiplication logic to handle both true and false cases (Example 6-13).

Example 6-13. Consolidating the log likelihood into a single line

```
log_likelihood = sum(log(likelihood)*p.y + log(1.0 - likelihood)*(1.0 - p.y) \
        for p in patient_data)
```

Finally, just plug these values in and get your R^2:

$$R^2 = \frac{(\text{log likelihood}) - (\text{log likelihood fit})}{(\text{log likelihood})}$$

$$R^2 = \frac{-0.5596 - (-9.9461)}{-0.5596} R^2 = 0.306456$$

And here is the Python code shown in Example 6-14, calculating the R^2 in its entirety.

Example 6-14. Calculating the R^2 for a logistic regression

```
import pandas as pd
from math import log, exp

patient_data = list(pd.read_csv('https://bit.ly/33ebs2R', delimiter=",") \
```

```
                                    .itertuples())

# Declare fitted logistic regression
b0 = -3.17576395
b1 = 0.69267212

def logistic_function(x):
    p = 1.0 / (1.0 + exp(-(b0 + b1 * x)))
    return p

# calculate the log likelihood of the fit
log_likelihood_fit = sum(log(logistic_function(p.x)) * p.y +
                         log(1.0 - logistic_function(p.x)) * (1.0 - p.y)
                         for p in patient_data)

# calculate the log likelihood without fit
likelihood = sum(p.y for p in patient_data) / len(patient_data)

log_likelihood = sum(log(likelihood) * p.y + log(1.0 - likelihood) * (1.0 - p.y) \
        for p in patient_data)

# calculate R-Square
r2 = (log_likelihood - log_likelihood_fit) / log_likelihood

print(r2)  # 0.306456105756576
```

OK, so we got an R^2 = 0.306456, so do hours of chemical exposure explain whether someone shows symptoms? As we learned in Chapter 5 on linear regression, a poor fit will be closer to an R^2 of 0.0 and a greater fit will be closer to 1.0. Therefore, we can conclude that hours of exposure is mediocre for predicting symptoms, as the R^2 is 0.30645. There must be variables other than time exposure that better predict if someone will show symptoms. This makes sense because we have a large mix of patients showing symptoms versus not showing symptoms for most of our observed data, as shown in Figure 6-12.

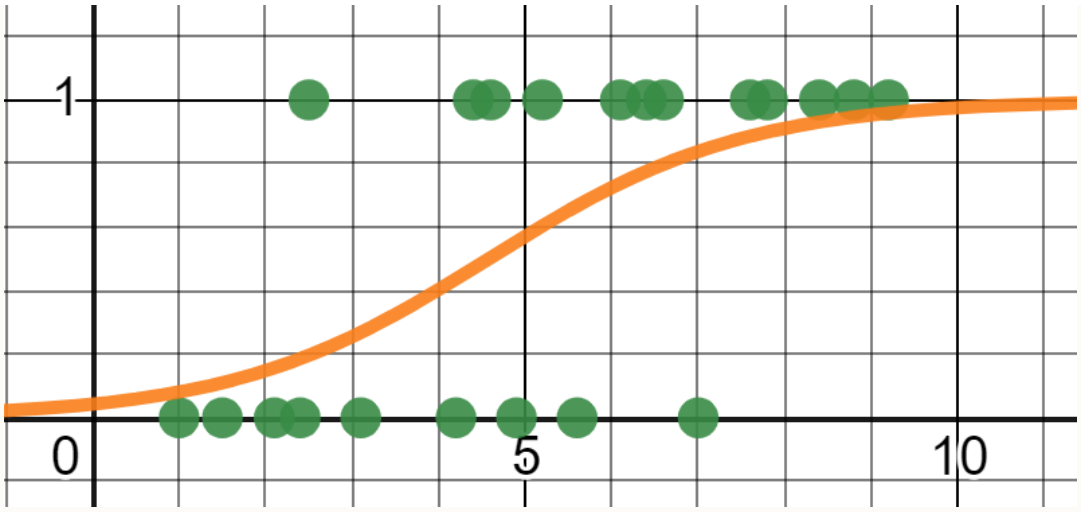


Figure 6-12. Our data has a mediocre R^2 of 0.30645 because there is a lot of variance in the middle of our curve

But if we did have a clean divide in our data, where 1 and 0 outcomes are cleanly separated as shown in Figure 6-13, we would have a perfect R^2 of 1.0.

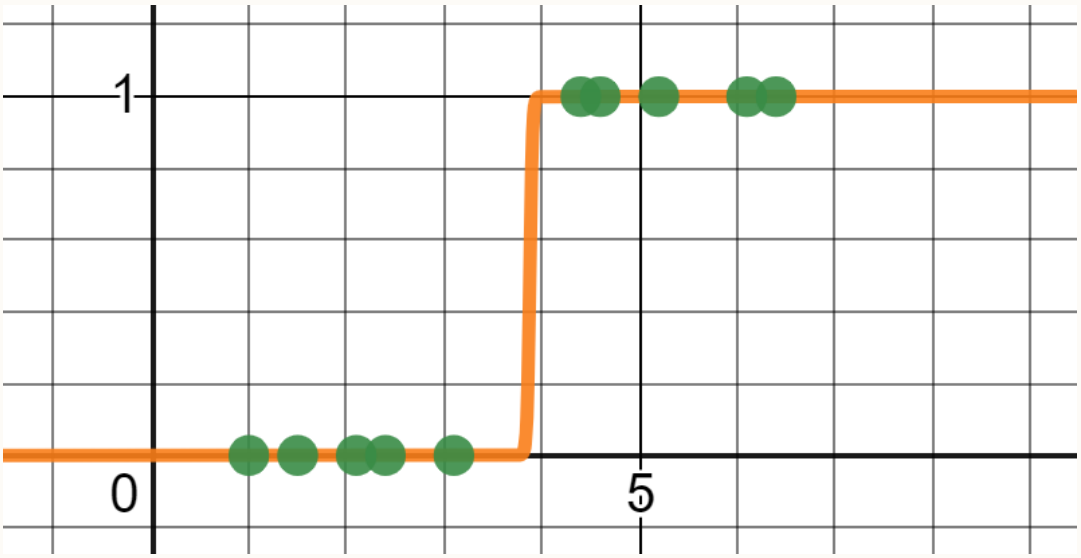


Figure 6-13. This logistic regression has a perfect R^2 of 1.0 because there is a clean divide in outcomes predicted by hours of exposure

P-Values

Just like linear regression, we are not done just because we have an R^2. We need to investigate how likely we would have seen this data by chance rather than because of an actual relationship. This means we need a p-value.

To do this, we will need to learn a new probability distribution called the *chi-square distribution*, annotated as χ^2 distribution. It is continuous and used in several areas of statistics, including this one!

If we take each value in a standard normal distribution (mean of 0 and standard deviation of 1) and square it, that will give us the χ^2 distribution with one degree of freedom. For our purposes, the degrees of freedom will depend on how many parameters n are in our logistic regression, which will be $n - 1$. You can see examples of different degrees of freedom in Figure 6-14.

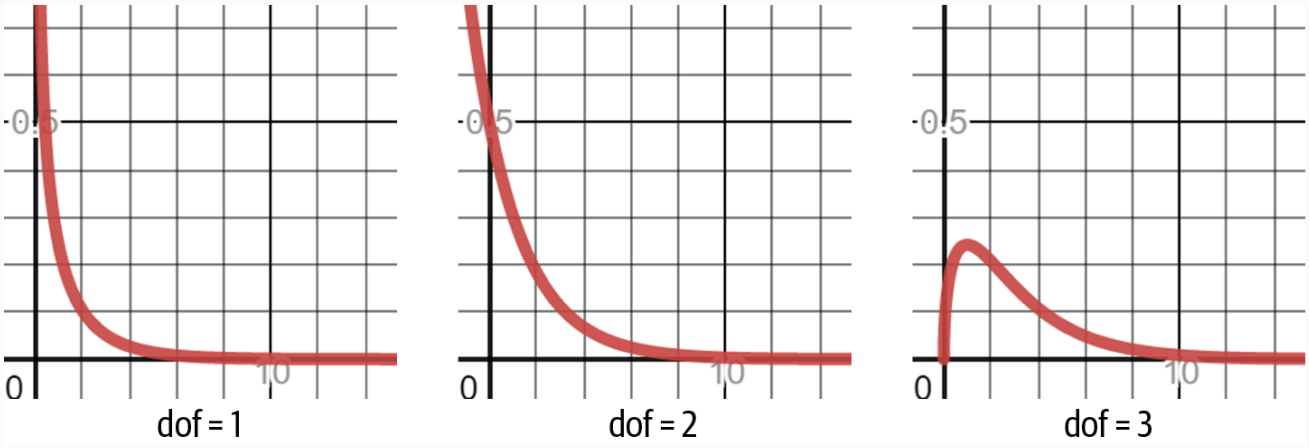


Figure 6-14. A χ^2 distribution with differing degrees of freedom

Since we have two parameters (hours of exposure and whether symptoms were shown), our degree of freedom will be 1 because $2 - 1 = 1$.

We will need the log likelihood fit and log likelihood as calculated in the previous subsection on R^2. Here is the formula that will produce the χ^2 value we need to look up:

$$\chi^2 = 2\,(\text{log likelihood fit}) - (\text{log likelihood})$$

We then take that value and look up the probability from the χ^2 distribution. That will give us our p-value:

$$\text{p-value} = \text{chi}(2((\text{log likelihood fit}) - (\text{log likelihood}))$$

Example 6-15 shows our p-value for a given fitted logistic regression. We use SciPy's `chi2` module to use the chi-square distribution.

Example 6-15. Calculating a p-value for a given logistic regression

```
import pandas as pd
from math import log, exp
from scipy.stats import chi2

patient_data = list(pd.read_csv('https://bit.ly/33ebs2R', delimiter=",").itertuples())

# Declare fitted logistic regression
b0 = -3.17576395
```

```
b1 = 0.69267212

def logistic_function(x):
    p = 1.0 / (1.0 + exp(-(b0 + b1 * x)))
    return p

# calculate the log likelihood of the fit
log_likelihood_fit = sum(log(logistic_function(p.x)) * p.y +
                         log(1.0 - logistic_function(p.x)) * (1.0 - p.y)
                         for p in patient_data)

# calculate the log likelihood without fit
likelihood = sum(p.y for p in patient_data) / len(patient_data)

log_likelihood = sum(log(likelihood) * p.y + log(1.0 - likelihood) * (1.0 - p.y) \
                     for p in patient_data)

# calculate p-value
chi2_input = 2 * (log_likelihood_fit - log_likelihood)
p_value = chi2.pdf(chi2_input, 1) # 1 degree of freedom (n - 1)

print(p_value)  # 0.0016604875618753787
```

So we have a p-value of 0.00166, and if our threshold for signifiance is .05, we say this data is statistically significant and was not by random chance.

Train/Test Splits

As covered in Chapter 5 on linear regression, we can use train/test splits as a way to validate machine learning algorithms. This is the more machine learning approach to assessing the performance of a logistic regression. While it is a good idea to rely on traditional statistical metrics like R^2 and p-values, when you are dealing with more variables, this becomes less practical. This is where train/test splits come in handy once again. To review, Figure 6-15 visualizes a three-fold cross-validation alternating a testing dataset.

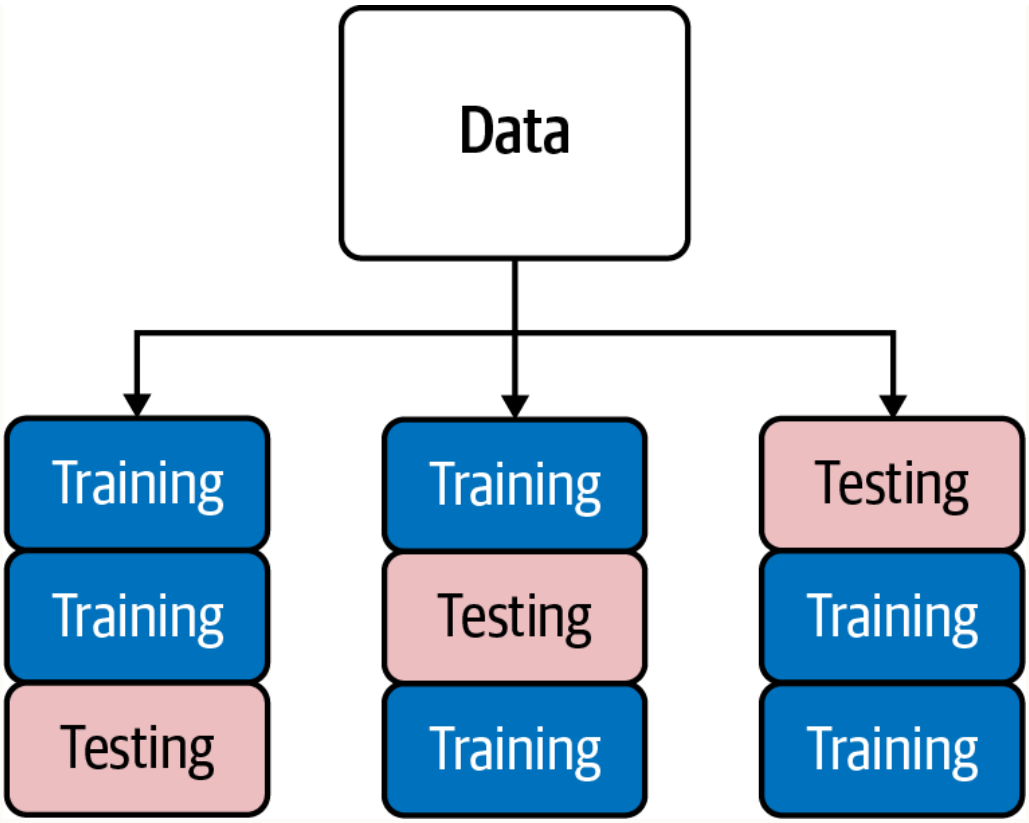


Figure 6-15. A three-fold cross-validation alternating each third of the dataset as a testing dataset

In Example 6-16 we perform a logistic regression on the employee-retention dataset, but we split the data into thirds. We then alternate each third as the testing data. Finally, we summarize the three accuracies with an average and standard deviation.

Example 6-16. Performing a logistic regression with three-fold cross-validation

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score

# Load the data
df = pd.read_csv("https://tinyurl.com/y6r7qjrp", delimiter=",")

X = df.values[:, :-1]
Y = df.values[:, -1]

# "random_state" is the random seed, which we fix to 7
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LogisticRegression(penalty='none')
results = cross_val_score(model, X, Y, cv=kfold)
```

```
print("Accuracy Mean: %.3f (stdev=%.3f)" % (results.mean(), results.std()))
```

We can also use random-fold validation, leave-one-out cross-validation, and all the other folding variants we performed in Chapter 5. With that out of the way, let's talk about why accuracy is a bad measure for classification.

Confusion Matrices

Suppose a model observed people with the name "Michael" quit their job. The reason why first and last names are captured as input variables is indeed questionable, as it is doubtful someone's name has any impact on whether they quit. However, to simplify the example, let's go with it. The model then predicts that any person named "Michael" will quit their job.

Now this is where accuracy falls apart. I have one hundred employees, including one named "Michael" and another named "Sam." Michael is wrongly predicted to quit, and it is Sam that ends up quitting. What's the accuracy of my model? It is 98% because there were only two wrong predictions out of one hundred employees as visualized in Figure 6-16.

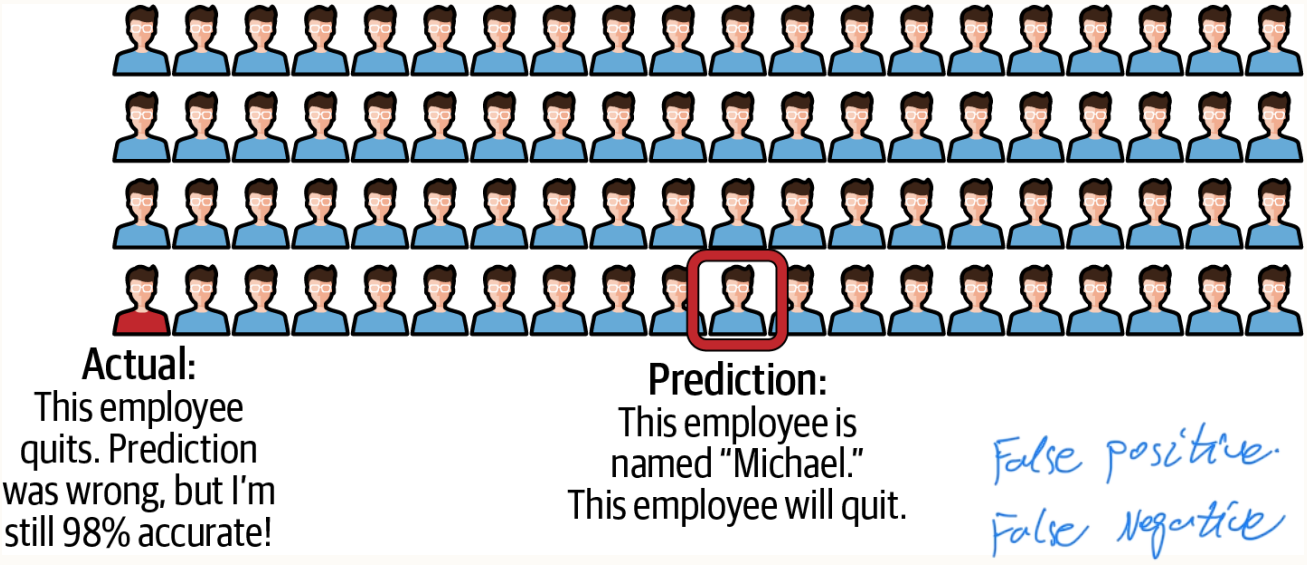


Figure 6-16. The employee named "Michael" is predicted to quit, but it's actually another employee that does, giving us 98% accuracy

Especially for imbalanced data where the event of interest (e.g., a quitting employee) is rare, the accuracy metric is horrendously misleading for classification problems. If a vendor, consultant, or data scientist ever tries to sell you a classification system on claims of accuracy, ask for a confusion matrix.

A *confusion matrix* is a grid that breaks out the predictions against the actual outcomes showing

the true positives, true negatives, false positives (type I error), and false negatives (type II error). Here is a confusion matrix presented in Figure 6-17.

	Actually quits (true)	Actually stays (false)
Predicted will quit (true)	0	1
Predicted will stay(false)	1	98

Figure 6-17. A simple confusion matrix

Generally, we want the diagonal values (top-left to bottom-right) to be higher because these reflect correct classifications. We want to evaluate how many employees who were predicted to quit actually did quit (true positives). Conversely, we also want to evaluate how many employees who were predicted to stay actually did stay (true negatives).

The other cells reflect wrong predictions, where an employee predicted to quit ended up staying (false positive), and where an employee predicted to stay ends up quitting (false negative).

What we need to do is dice up that accuracy metric into more specific accuracy metrics targeting different parts of the confusion matrix. Let's look at Figure 6-18, which adds some useful measures.

From the confusion matrix, we can derive all sorts of useful metrics beyond just accuracy. We can easily see that precision (how accurate positive predictions were) and sensitivity (rate of identified positives) are 0, meaning this machine learning model fails entirely at positive predictions.

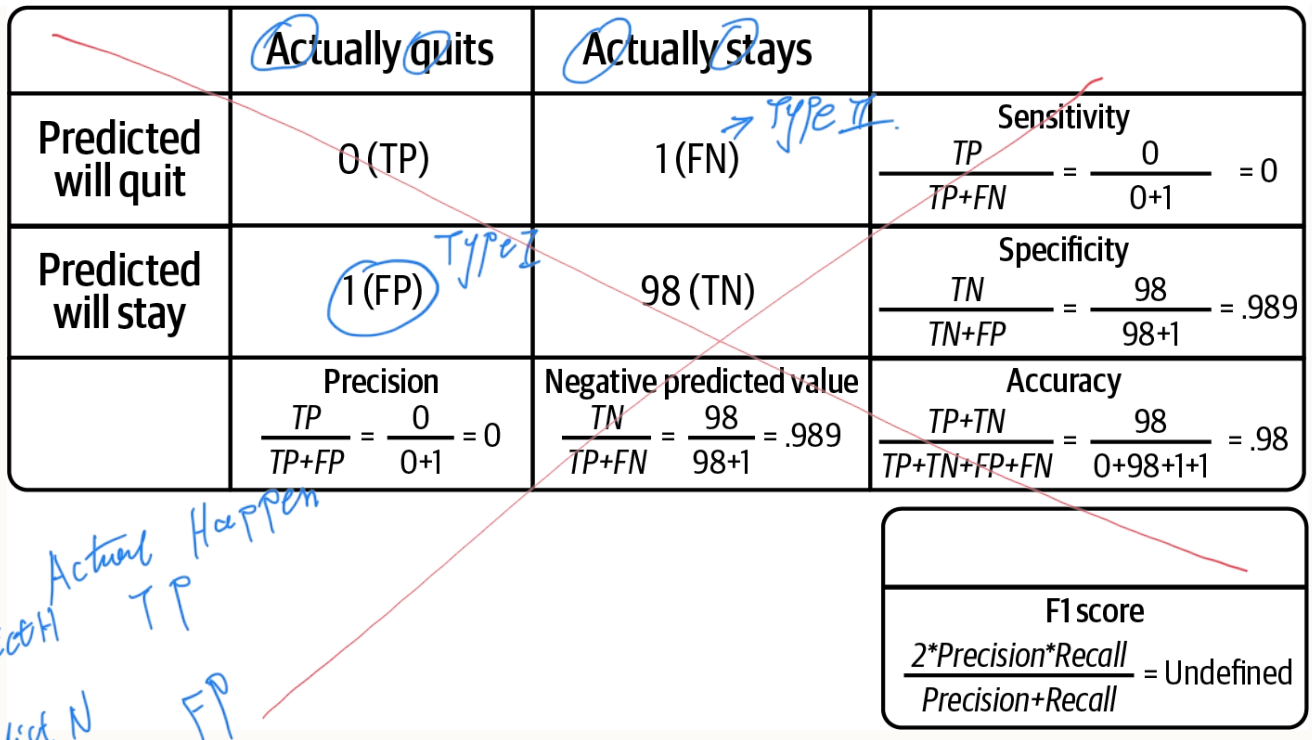


Figure 6-18. Adding useful metrics to the confusion matrix

Example 6-17 shows how to use the confusion matrix API in SciPy on a logistic regression with a train/test split. Note that the confusion matrix is only applied to the testing dataset.

Example 6-17. Creating a confusion matrix for a testing dataset in SciPy

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

# Load the data
df = pd.read_csv('https://bit.ly/3cManTi', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

model = LogisticRegression(solver='liblinear')

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.33,
    random_state=10)
model.fit(X_train, Y_train)
```

```
prediction = model.predict(X_test)

"""
The confusion matrix evaluates accuracy within each category.
[[truepositives falsenegatives]
 [falsepositives truenegatives]]

The diagonal represents correct predictions,
so we want those to be higher
"""
matrix = confusion_matrix(y_true=Y_test, y_pred=prediction)
print(matrix)
```

Bayes' Theorem and Classification

Do you recall Bayes' Theorem in Chapter 2? You can use Bayes' Theorem to bring in outside information to further validate findings on a confusion matrix. Figure 6-19 shows a confusion matrix of one thousand patients tested for a disease.

	Tests positive	Tests negative
At risk	198	2
Not at risk	50	750

Figure 6-19. A confusion matrix for a medical test identifying a disease

We are told that for patients that have a health risk, 99% will be identified successfully (sensitivity). Using the confusion matrix, we can see this mathematically checks out:

$$\text{sensitivity} = \frac{198}{198+2} = .99$$

But what if we flip the condition? What percentage of those who tested positive have the health risk (precision)? While we are flipping a conditional probability, we do not have to use Bayes' Theorem here because the confusion matrix gives us all the numbers we need:

$$\text{precision} = \frac{198}{198+50} = .798$$

OK, so 79.8% is not terrible, and that's the percentage of people who tested positive that actually have the disease. But ask yourself this…what are we assuming about our data? Is it representative of the population?

Some quick research found 1% of the population actually has the disease. There is an opportunity to use Bayes' Theorem here. We can account for the proportion of the population that actually has the disease and incorporate that into our confusion matrix findings. We then discover something significant.

$$P(\text{At Risk if Positive}) = \frac{P(\text{Positive if At Risk}) \times P(\text{At Risk})}{P(\text{Positive})}$$

$$P(\text{At Risk if Positive}) = \frac{.99 \times .01}{.248}$$

$$P(\text{At Risk if Positive}) = .0339$$

When we account for the fact that only 1% of the population is at risk, and 20% of our test patients are at risk, the probability of being at risk given a positive test is 3.39%! How did it drop from 99%? This just shows how easily we can get duped by probabilities that are high only in a specific sample like the vendor's one thousand test patients. So if this test has only a 3.39% probability of successfully identifying a true positive, we probably should not use it.

AUC : Area Under Curve

Receiver Operator Characteristics/Area Under Curve

ROC : receiver operator characteristics

When we are evaluating different machine learning configurations, we may end up with dozens, hundreds, or thousands of confusion matrices. These can be tedious to review, so we can summarize all of them with a *receiver operator characteristic (ROC) curve* as shown in Figure 6-20. This allows us to see each testing instance (each represented by a black dot) and find an agreeable balance between true positives and false positives.

We can also compare different machine learning models by creating separate ROC curves for each. For example, if in Figure 6-21 our top curve represents a logistic regression and the bottom curve represents a decision tree (a machine learning technique we did not cover in this book), we can see the performance of them side by side. The *area under the curve (AUC)* is a good metric for choosing which model to use. Since the top curve (logistic regression) has a greater area, this suggests it is a superior model.

AUC

ROC

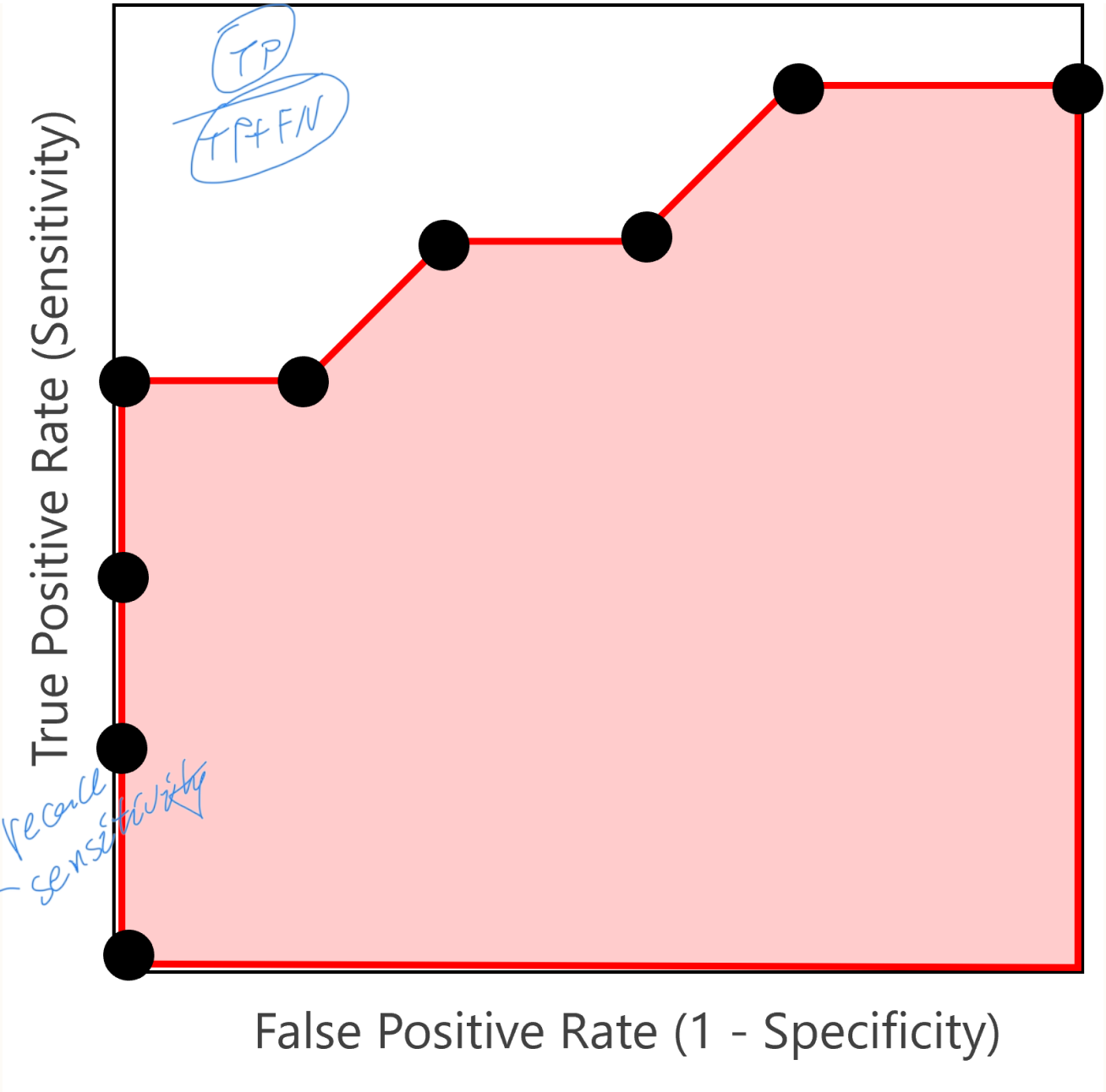


Figure 6-20. A receiver operator characteristic curve

$$\frac{TP}{TP + FP}$$

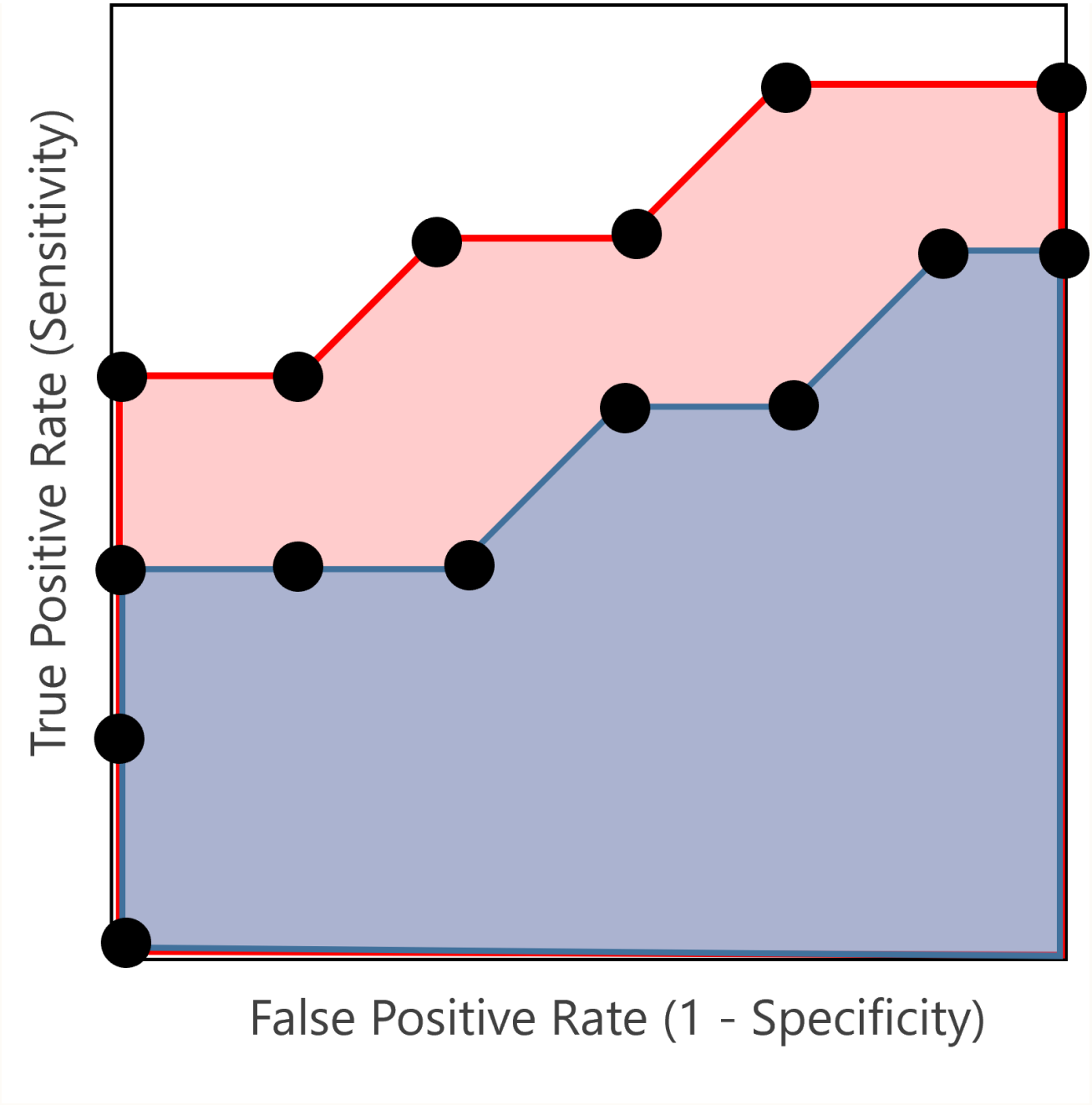


Figure 6-21. Comparing two models by their area under the curve (AUC) with their respective ROC curves

To use the AUC as a scoring metric, change the `scoring` parameter in the scikit-learn API to use `roc_auc` as shown for a cross-validation in Example 6-18.

Example 6-18. Using the AUC as the scikit-learn parameter

```
# put Scikit_learn model here

results = cross_val_score(model, X, Y, cv=kfold, scoring='roc_auc')
print("AUC: %.3f (%.3f)" % (results.mean(), results.std()))
```

```
# AUC: 0.791 (0.051)
```

Class Imbalance

There is one last thing to cover before we close this chapter. As we saw earlier when discussing confusion matrices, *class imbalance*, which happens when data is not equally represented across every outcome class, is a problem in machine learning. Unfortunately, many problems of interest are imbalanced, such as disease prediction, security breaches, fraud detection, and so on. Class imbalance is still an open problem with no great solution. However, there are a few techniques you can try.

First, you can do obvious things like collect more data or try different models as well as use confusion matrices and ROC/AUC curves. All of this will help track poor predictions and proactively catch errors.

Another common technique is to duplicate samples in the minority class until it is equally represented in the dataset. You can do this in scikit-learn as shown in Example 6-19 when doing your train-test splits. Pass the `stratify` option with the column containing the class values, and it will attempt to equally represent the data of each class.

Example 6-19. Using the `stratify` option in scikit-learn to balance classes in the data

```
X, Y = ...
X_train, X_test, Y_train, Y_test = \
        train_test_split(X, Y, test_size=.33, stratify=Y)
```

There is also a family of algorithms called SMOTE, which generate synthetic samples of the minority class. What would be most ideal though is to tackle the problem in a way that uses anomaly-detection models, which are deliberately designed for seeking out a rare event. These seek outliers, however, and are not necessarily a classification since they are unsupervised algorithms. All these techniques are beyond the scope of this book but are worth mentioning as they *might* provide better solutions to a given problem.

Conclusion

Logistic regression is the workhorse model for predicting probabilities and classifications on data. Logistic regressions can predict more than one category rather than just a true/false. You

just build separate logistic regressions modeling whether or not it belongs to that category, and the model that produces the highest probability is the one that wins. You may discover that scikit-learn, for the most part, will do this for you and detect when your data has more than two classes.

In this chapter, we covered not just how to fit a logistic regression using gradient descent and scikit-learn but also statistical and machine learning approaches to validation. On the statistical front we covered the R^2 and p-value, and in machine learning we explored train/test splits, confusion matrices, and ROC/AUC.

If you want to learn more about logistic regression, probably the best resource to jump-start further is Josh Starmer's StatQuest playlist on Logistic Regression. I have to credit Josh's work in assisting some portions of this chapter, particularly in how to calculate R^2 and p-values for logistic regression. If nothing else, watch his videos for the fantastic opening jingles!

As always, you will find yourself walking between the two worlds of statistics and machine learning. Many books and resources right now cover logistic regression from a machine learning perspective, but try to seek out statistics resources too. There are advantages and disadvantages to both schools of thought, and you can win only by being adaptable to both!

Exercises

A dataset of three input variables `RED`, `GREEN`, and `BLUE` as well as an output variable `LIGHT_OR_DARK_FONT_IND` is provided here. It will be used to predict whether a light/dark font (0/1 respectively) will work for a given background color (specified by RGB values).

1. Perform a logistic regression on the preceding data, using three-fold cross-validation and accuracy as your metric.
2. Produce a confusion matrix comparing the predictions and actual data.
3. Pick a few different background colors (you can use an RGB tool like this one) and see if the logistic regression sensibly chooses a light (0) or dark (1) font for each one.
4. Based on the preceding exercises, do you think logistic regression is effective for predicting a light or dark font for a given background color?

Answers are in Appendix B.