**Introduction**

When an email lands in your inbox, how does your email service know whether it's a real email or spam? This evaluation is made billions of times per day, and one way it can be done is with Logistic Regression. ***Logistic Regression*** is a supervised machine learning algorithm that uses regression to predict the continuous probability, ranging from 0 to 1, of a data sample belonging to a specific category, or class. Then, based on that probability, the sample is classified as belonging to the more probable class, ultimately making Logistic Regression a classification algorithm.

In our spam filtering example, a Logistic Regression model would predict the probability of an incoming email being spam. If that predicted probability is greater than or equal to 0.5, the email is classified as spam. We would call spam the *positive class*, with the label 1, since the positive class is the class our model is looking to detect. If the predicted probability is less than 0.5, the email is classified as ham (a real email). We would call ham the *negative class*, with the label 0. This act of deciding which of two classes a data sample belongs to is called *binary classification*.

Some other examples of what we can classify with Logistic Regression include:

* Disease survival —Will a patient, 5 years after treatment for a disease, still be alive?
* Customer conversion —Will a customer arriving on a sign-up page enroll in a service?

In this lesson you will learn how to perform Logistic Regression and use it to make classifications on your own data!

If you are unfamiliar with Linear Regression, we recommend you go check out our [Linear Regression course](https://www.codecademy.com/paths/data-science/tracks/dspath-supervised/modules/dspath-linear-regression/lessons/linear-regression/exercises/introduction) before proceeding to Logistic Regression. If you are familiar, let's dive in!

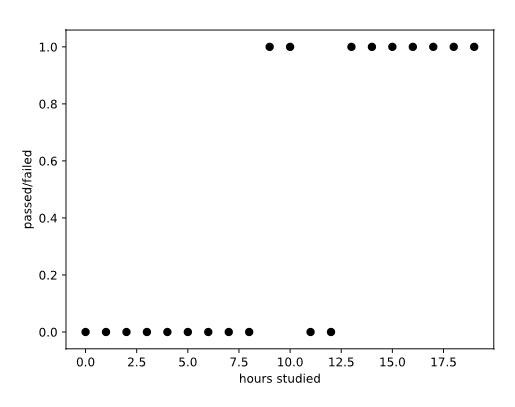
Instructions

**1.**

Codecademy University's Data Science department is interested in creating a model to predict whether or not a student will pass the final exam of its Introductory Machine Learning course. The department thinks a Logistic Regression model that makes predictions based on the number of hours a student studies will work well. To aid the investigation, the department asked a supplemental question on the exam: how many hours did you study?

Run the code in **script.py** to plot the data samples. 0 indicates that a student failed the exam, and 1 indicates a student passed the exam.   
  
How many hours does a student need to study to pass the exam?

import codecademylib3\_seaborn  
import numpy as np  
import matplotlib.pyplot as plt  
from exam import hours\_studied, passed\_exam, math\_courses\_taken  
  
# Scatter plot of exam passage vs number of hours studied  
plt.scatter(hours\_studied.ravel(), passed\_exam, color='black', zorder=20)  
plt.ylabel('passed/failed')  
plt.xlabel('hours studied')  
  
plt.show()



**Linear Regression Approach**

With the data from Codecademy University, we want to predict whether each student will pass their final exam. And the first step to making that prediction is to predict the probability of each student passing. Why not use a **Linear Regression** model for the prediction, you might ask? Let's give it a try.

Recall that in Linear Regression, we fit a regression line of the following form to the data:

y=b0+b1x1+b2x2+⋯+bnxn

where

* y is the value we are trying to predict
* b\_0 is the intercept of the regression line
* b\_1, b\_2, … b\_n are the coefficients of the features x\_1, x\_2, … x\_n of the regression line

For our data points y is either 1 (passing), or 0 (failing), and we have one feature, num\_hours\_studied. Below we fit a Linear Regression model to our data and plotted the results, with the line of best fit in red.

A problem quickly arises. For low values of num\_hours\_studied the regression line predicts negative probabilities of passing, and for high values of num\_hours\_studied the regression line predicts probabilities of passing greater than 1. These probabilities are meaningless! We get these meaningless probabilities since the output of a Linear Regression model ranges from -∞ to +∞.

Instructions

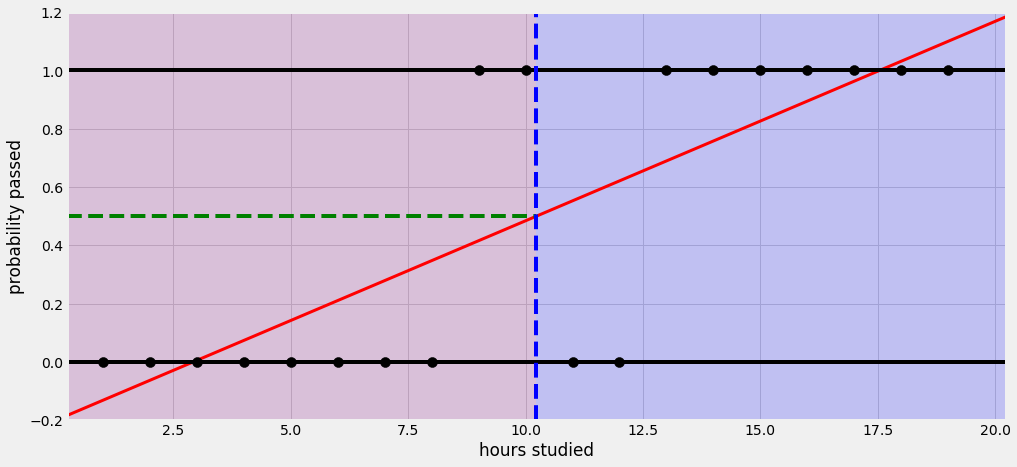
**1.**

Provided to you is the code to train a linear regression model on the Codecademy University data and plot the regression line. Run the code and observe the plot. Expand the plot to fullscreen for a larger view.

Using the regression line, estimate the probability of passing for a student who studies 1 hour and for a student who studies 19 hours. Save the results to slacker and studious, respectively.

What is wrong with using a Linear Regression model to predict these probabilities?

import codecademylib3\_seaborn  
import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.linear\_model import LinearRegression  
from exam import hours\_studied, passed\_exam  
from plotter import plot\_data  
  
# Create linear regression model  
model = LinearRegression()  
model.fit(hours\_studied,passed\_exam)  
  
# Plug sample data into fitted model  
sample\_x = np.linspace(-16.65, 33.35, 300).reshape(-1,1)  
probability = model.predict(sample\_x).ravel()  
  
# Function to plot exam data and linear regression curve  
plot\_data(model)  
  
# Show the plot  
plt.show()  
  
# Define studious and slacker here  
studious = 1.1  
slacker = -0.1



**Logistic Regression**

We saw that the output of a Linear Regression model does not provide the probabilities we need to predict whether a student passes the final exam. Step in ***Logistic Regression!***

In Logistic Regression we are also looking to find coefficients for our features, but this time we are fitting a logistic curve to the data so that we can predict probabilities. Described below is an overview of how Logistic Regression works. Don't worry if something does not make complete sense right away, we will dig into each of these steps in further detail in the remaining exercises!

To predict the probability of a data sample belonging to a class, we:

1. initialize all feature coefficients and intercept to 0
2. multiply each of the feature coefficients by their respective feature value to get what is known as the *log-odds*
3. place the log-odds into the *sigmoid function* to link the output to the range [0,1], giving us a probability

By comparing the predicted probabilities to the actual classes of our data points, we can evaluate how well our model makes predictions and use [*gradient descent*](https://www.codecademy.com/paths/data-science/tracks/dspath-supervised/modules/dspath-linear-regression/lessons/linear-regression/exercises/gradient-descent-b) to update the coefficients and find the best ones for our model.

To then make a final classification, we use a *classification threshold* to determine whether the data sample belongs to the positive class or the negative class.

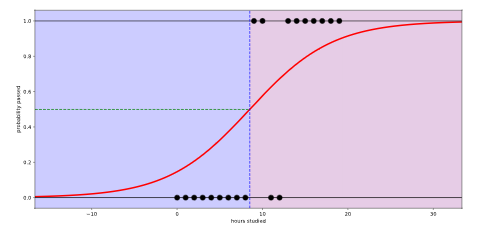
Instructions

**1.**

Provided to you is the code to build a Logistic Regression model on the Codecademy University data and plot the fitted logistic curve. Run the code and observe the plot. Expand the plot to fullscreen for a larger view.

What is the lowest possible probability that can be predicted, and what is the highest possible probability that can be predicted? Enter your answer in the variables lowest and highest, respectively.

import codecademylib3\_seaborn  
import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.linear\_model import LogisticRegression  
from exam import hours\_studied, passed\_exam  
from plotter import plot\_data  
  
# Create logistic regression model  
model = LogisticRegression()  
model.fit(hours\_studied,passed\_exam)  
  
# Plug sample data into fitted model  
sample\_x = np.linspace(-16.65, 33.35, 300).reshape(-1,1)  
probability = model.predict\_proba(sample\_x)[:,1]  
  
# Function to plot exam data and logistic regression curve  
plot\_data(model)  
  
# Show the plot  
plt.show()  
  
# Lowest and highest probabilities  
lowest = 0.0  
highest = 1.0



**Log-Odds**

In Linear Regression we multiply the coefficients of our features by their respective feature values and add the intercept, resulting in our prediction, which can range from -∞ to +∞. In Logistic Regression, we make the same multiplication of feature coefficients and feature values and add the intercept, but instead of the prediction, we get what is called the ***log-odds***.

The log-odds are another way of expressing the probability of a sample belonging to the positive class, or a student passing the exam. In probability, we calculate the *odds* of an event occurring as follows:

Odds = P(event occurring) / P(event not occurring)

The odds tell us how many more times likely an event is to occur than not occur. If a student will pass the exam with probability 0.7, they will fail with probability 1 - 0.7 = 0.3. We can then calculate the odds of passing as:

Odds of passing = 0.7 / 0.3 = 2.33‾ (3‾ = 3s-repeating)

The log-odds are then understood as the logarithm of the odds!

Log odds of passing = log(2.33‾) = 0.847

For our Logistic Regression model, however, we calculate the log-odds, represented by z below, by summing the product of each feature value by its respective coefficient and adding the intercept. This allows us to map our feature values to a measure of how likely it is that a data sample belongs to the positive class.

z = b0+b1x1+⋯+bnxn

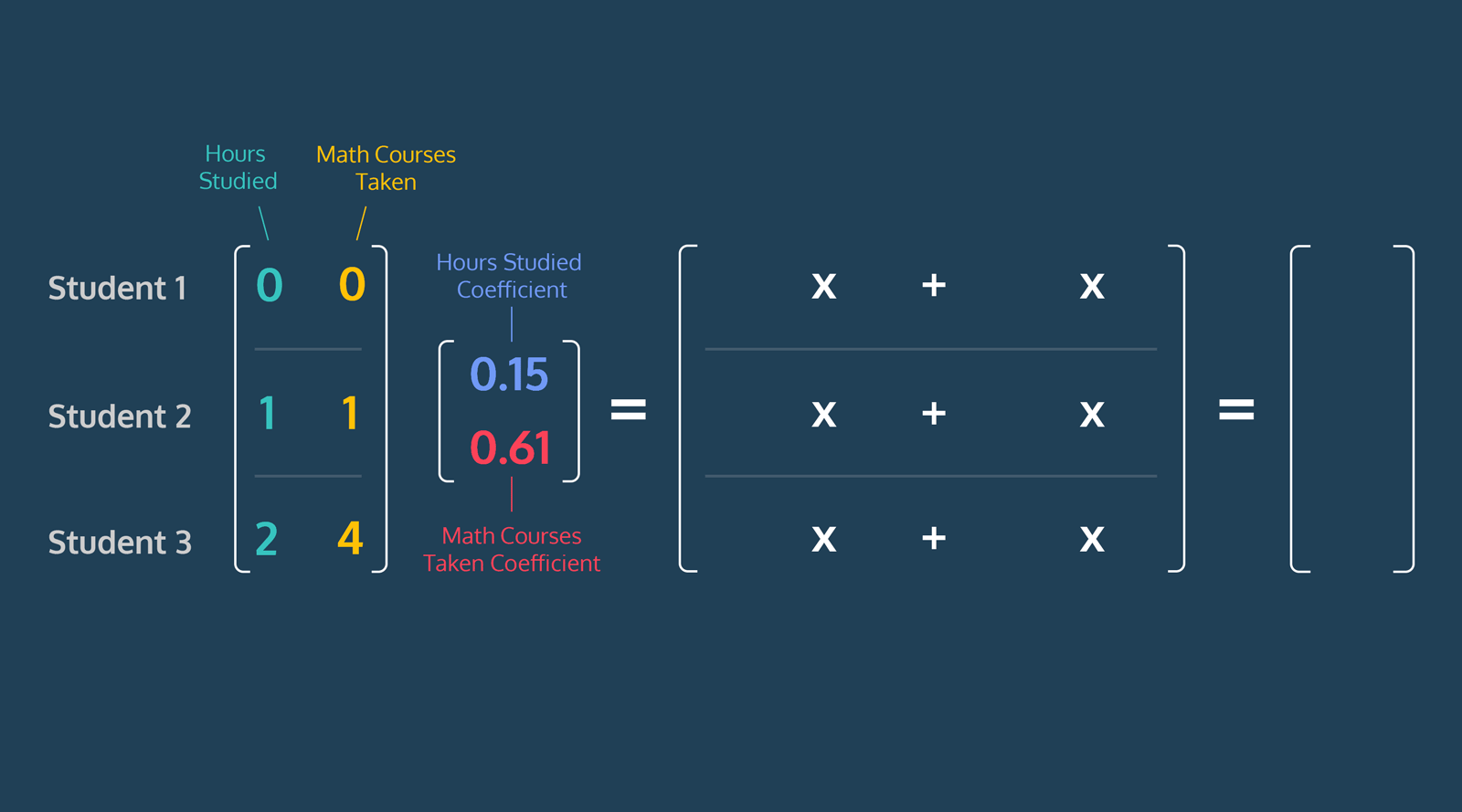
* b\_0 is the intercept
* b\_1, b\_2, … b\_n are the coefficients of the features x\_1, x\_2, … x\_n

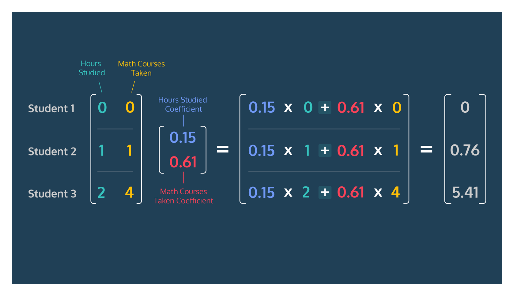
This kind of multiplication and summing is known as a *dot product*.

We can perform a dot product using numpy's np.dot() method! Given feature matrix features, coefficient vector coefficients, and an intercept, we can calculate the log-odds in numpy as follows:

log\_odds = np.dot(features, coefficients) + intercept

np.dot() will take each row, or student, in features and multiply each individual feature value by its respective coefficient in coefficients, summing the result, as shown below.





We then add in the intercept to get the log-odds!

**1.**

Let's create a function log\_odds that takes features, coefficients and intercept as parameters. For now return features.

**2.**

Update log\_odds to return the dot product of features and coefficients.

**3.**

Update the return statement of log-odds by adding the intercept after the dot product.

**4.**

With the log\_odds function you created, let's calculate the log-odds of passing for the Introductory Machine Learning students. Use hours\_studied as the features, calculated\_coefficients as the coefficients and intercept as the intercept. Store the result in calculated\_log\_odds, and print it out.

import numpy as np  
from exam import hours\_studied, calculated\_coefficients, intercept  
  
# Create your log\_odds() function here  
def log\_odds(features, coefficients, intercept):  
 return np.dot(features, coefficients)+intercept  
  
# Calculate the log-odds for the Codecademy University data here  
calculated\_log\_odds = log\_odds(hours\_studied, calculated\_coefficients, intercept)  
print(calculated\_log\_odds)

[[-1.76125712]

[-1.55447221]

[-1.3476873 ]

[-1.14090239]

[-0.93411748]

[-0.72733257]

[-0.52054766]

[-0.31376275]

[-0.10697784]

[ 0.09980707]

[ 0.30659198]

[ 0.51337689]

[ 0.7201618 ]

[ 0.92694671]

[ 1.13373162]

[ 1.34051653]

[ 1.54730144]

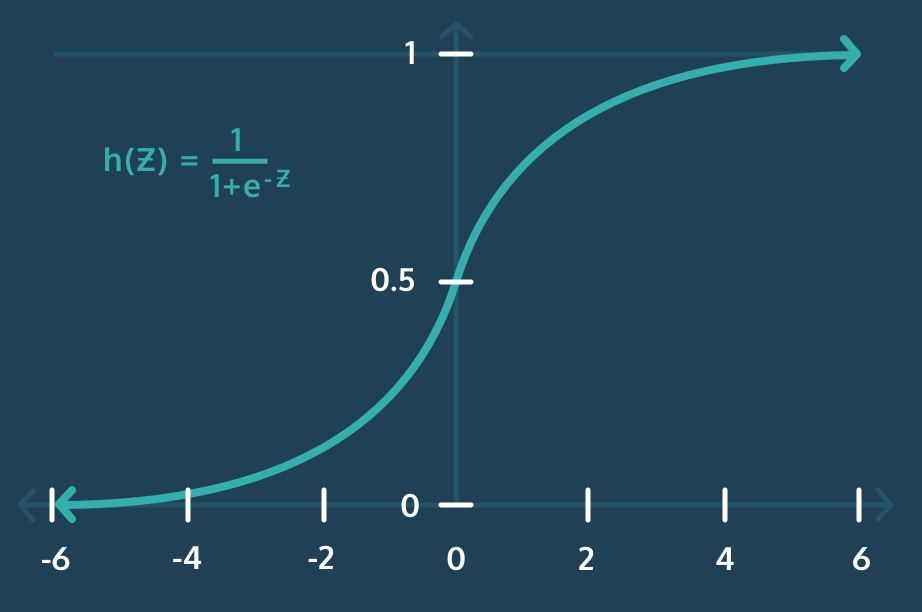
[ 1.75408635]

[ 1.96087126]

[ 2.16765617]]

**Sigmoid Function**

How did our Logistic Regression model create the S-shaped curve we previously saw? The answer is the ***Sigmoid Function***.



The Sigmoid Function is a special case of the more general *Logistic Function*, where Logistic Regression gets its name. Why is the Sigmoid Function so important? By plugging the log-odds into the Sigmoid Function, defined below, we map the log-odds z to the range [0,1].

h(z) = 1 / (1+e^(−z))

* e^(-z) is the exponential function, which can be written in numpy as np.exp(-z)

This enables our Logistic Regression model to output the probability of a sample belonging to the positive class, or in our case, a student passing the final exam!

**1.**

Let's create a Sigmoid Function of our own! Define a function called sigmoid() that takes z as a parameter. For now, have it return z.

**2.**

Inside the function and above the return statement, create a variable denominator and set it equal to 1 plus the exponential of -z. Instead of returning z, return 1/denominator.

**3.**

All done! Now test out your function by plugging in the calculated\_log\_odds we found in the previous exercise and saving the result to probabilities. Then, print probabilities.

import codecademylib3\_seaborn  
import numpy as np  
from exam import calculated\_log\_odds  
  
# Create your sigmoid function here  
def sigmoid(z):  
 denominator = 1 + np.exp(-z)  
 return 1/denominator  
  
# Calculate the sigmoid of the log-odds here  
probabilities = sigmoid(calculated\_log\_odds)  
print(probabilities)

[[0.14663296]

[0.17444128]

[0.20624873]

[0.24215472]

[0.28209011]

[0.32578035]

[0.37272418]

[0.42219656]

[0.47328102]

[0.52493108]

[0.57605318]

[0.62559776]

[0.67264265]

[0.71645543]

[0.7565269 ]

[0.79257487]

[0.82452363]

[0.85246747]

[0.87662721]

[0.89730719]]

**Log-Loss I**

Now that we understand how a Logistic Regression model makes its probability predictions, what coefficients and intercept should we use in our model to best predict whether a student will pass the exam? To answer this question we need a way to evaluate how well a given model fits the data we have.

The function used to evaluate the performance of a machine learning model is called a *loss function*, or a *cost function*. To evaluate how "good a fit" a model is, we calculate the loss for each data sample (how wrong the model's prediction was) and then average the loss across all samples. The loss function for Logistic Regression, known as **Log Loss**, is given below:

−1m∑i=1m[y(i)log(h(z(i)))+(1−y(i))log(1−h(z(i)))]-\frac{1}{m}\sum\_{i=1}^{m} [y^{(i)}log(h(z^{(i)})) + (1-y^{(i)})log(1-h(z^{(i)}))]−m

1​i=1∑m​[y(i)log(h(z(i)))+(1−y(i))log(1−h(z(i)))]

* m is the total number of data samples
* y\_i is the class of data sample i
* z\_i is the log-odds of sample i
* h(z\_i) is the sigmoid of the log-odds of sample i, which is the probability of sample i belonging to the positive class

The log-loss function might seem scary, but don't worry, we are going to break it down in the next exercise!

The goal of our Logistic Regression model is to find the feature coefficients and intercept, which shape the logistic function, that minimize log-loss for our training data!

Instructions

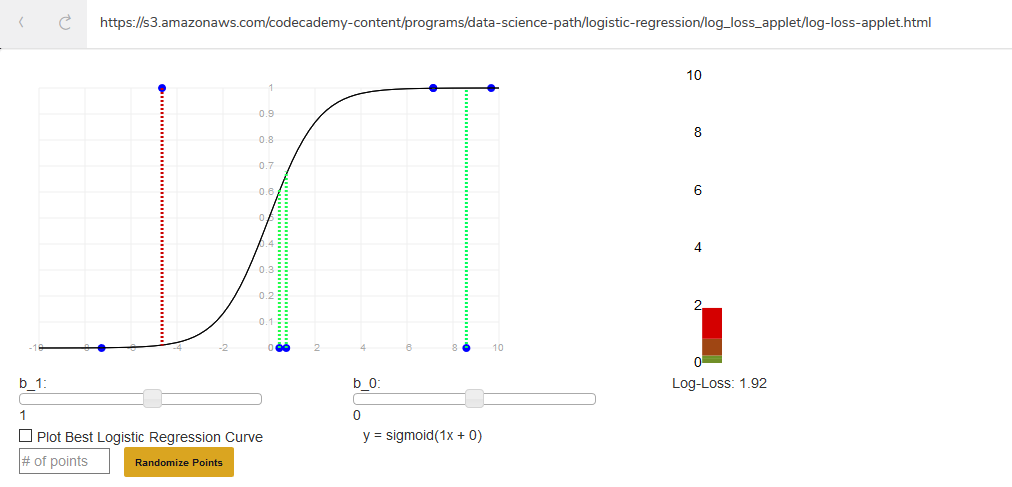
The interactive visualization in the browser lets you try to find the logistic curve that minimizes log-loss for a randomized set of data points with one feature:

* The slider on the left controls b\_1 (coefficient of feature 1)
* The slider on the right controls b\_0 (intercept)
* You can see the log-loss on the right side of the visualization.

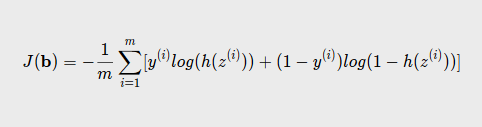
To check if your logistic curve minimizes the log-loss, check the "Plot Best Logistic Regression Curve" box.

Randomize a new set of samples and try to fit a new logistic curve by entering the number of samples you want (try 8!) in the text box and pressing Randomize Points.

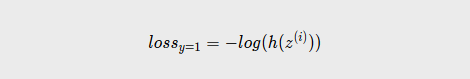
<https://s3.amazonaws.com/codecademy-content/programs/data-science-path/logistic-regression/log_loss_applet/log-loss-applet.html>



# Log Loss II

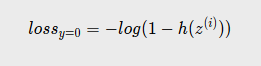


Let's go ahead and break down our log-loss function into two separate parts so it begins to make more sense. Consider the case when a data sample has class y = 1, or for our data when a student passed the exam. The right-side of the equation drops out because we end up with 1 - 1 (or 0) multiplied by some value. The loss for that individual student becomes:



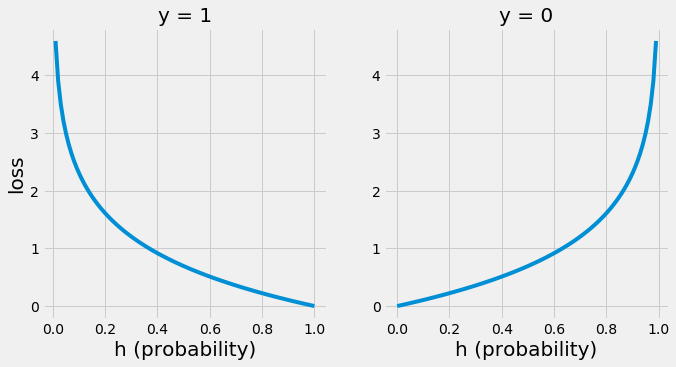
The loss for a student who passed the exam is just the log of the probability the student passed the exam!

And for a student who fails the exam, where a sample has class y = 0, the left-side of the equation drops out and the loss for that student becomes:



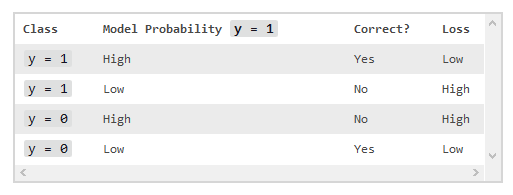
The loss for a student who failed the exam is the log of one minus the probability the student passed the exam, which is just the log of the probability the student failed the exam!

Let's take a closer look at what is going on with our loss function by graphing the loss of individual samples when the class label is y = 1 and y = 0.



Let's go back to our Codecademy University data and consider four possible cases:

| **Class** | **Model Probability y = 1** | **Correct?** | **Loss** |
| --- | --- | --- | --- |
| y = 1 | High | Yes | Low |
| y = 1 | Low | No | High |
| y = 0 | High | No | High |
| y = 0 | Low | Yes | Low |



From the graphs and the table you can see that confident correct predictions result in small losses, while confident incorrect predictions result in large losses that approach infinity. This makes sense! We want to punish our model with an increasing loss as it makes progressively incorrect predictions, and we want to reward the model with a small loss as it makes correct predictions.

Just like in Linear Regression, we can then use gradient descent to find the coefficients that minimize log-loss across all of our training data.

**1.**

Let's calculate the log-loss for our Codecademy University data. To calculate loss we need the actual classes, pass (1), or fail (0), for the students. Print passed\_exam to inspect the actual classes.

**2.**

In the code editor, we've provided you with a function log\_loss() that calculates the log-loss for a set of predicted probabilities and their actual classes. Use probabilities, which you calculated previously, and passed\_exam as inputs to log\_loss() and store the result in loss\_1. Print loss\_1.

**3.**

Now that we have calculated the loss for our best coefficients, let's compare this loss to the loss we begin with when we initialize our coefficients and intercept to 0. probabilities\_2 contains the calculated probabilities of the students passing the exam with the coefficient for hours\_studied set to 0. Use probabilities\_2 and passed\_exam as inputs to log\_loss() and store the result in loss\_2. Print loss\_2.

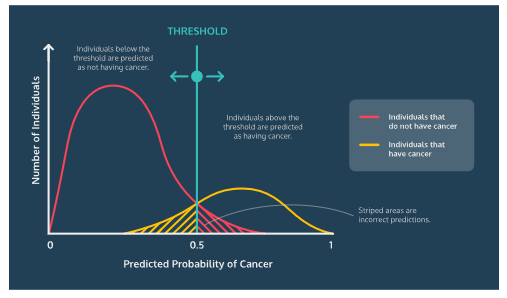
Which set of coefficients produced the lower log-loss?

import numpy as np  
from exam import passed\_exam, probabilities, probabilities\_2  
  
# Function to calculate log-loss  
def log\_loss(probabilities,actual\_class):  
 return np.sum(-(1/actual\_class.shape[0])\*(actual\_class\*np.log(probabilities) + (1-actual\_class)\*np.log(1-probabilities)))  
  
# Print passed\_exam here  
print(passed\_exam)  
  
# Calculate and print loss\_1 here  
loss\_1 = log\_loss(probabilities, passed\_exam)  
print(loss\_1) # 0.398640332141742  
  
# Calculate and print loss\_2 here  
loss\_2 = log\_loss(probabilities\_2, passed\_exam)  
print(loss\_2) # 13.862943611198906  
  
# the first set of probabilities produced a lower log-loss

# Classification Thresholding

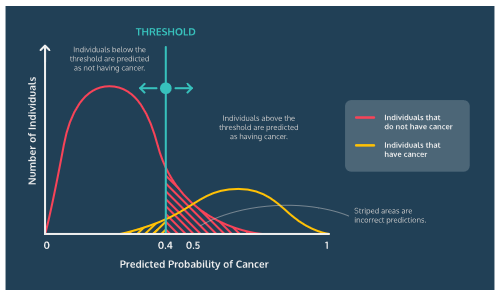
Many machine learning algorithms, including Logistic Regression, spit out a classification probability as their result. Once we have this probability, we need to make a decision on what class the sample belongs to. This is where the ***classification threshold*** comes in!

The default threshold for many algorithms is 0.5. If the predicted probability of an observation belonging to the positive class is greater than or equal to the threshold, 0.5, the classification of the sample is the positive class. If the predicted probability of an observation belonging to the positive class is less than the threshold, 0.5, the classification of the sample is the negative class.



We can choose to change the threshold of classification based on the use-case of our model. For example, if we are creating a Logistic Regression model that classifies whether or not an individual has cancer, we want to be more sensitive to the positive cases, signifying the presence of cancer, than the negative cases.

In order to ensure that most patients with cancer are identified, we can move the classification threshold down to 0.3 or 0.4, increasing the sensitivity of our model to predicting a positive cancer classification. While this might result in more overall misclassifications, we are now missing fewer of the cases we are trying to detect: actual cancer patients.



**1.**

Let's use all the knowledge we've gathered to create a function that performs thresholding and makes class predictions! Define a function predict\_class() that takes a features matrix, a coefficients vector, an intercept, and a threshold as parameters. Return threshold.

**2.**

In predict\_class(), calculate the log-odds using the log\_odds() function we defined earlier. Store the result in calculated\_log\_odds, and return calculated\_log\_odds.

**3.**

Still in predict\_class(), find the probabilities that the samples belong to the positive class. Create a variable probabilities, and give it the value returned by calling sigmoid() on calculated\_log\_odds. Return probabilities.

**4.**

Return 1 for all values within probabilities equal to or above threshold, and 0 for all values below threshold.

**5.**

Let's make final classifications on our Codecademy University data to see which students passed the exam. Use the predict\_class() function with hours\_studied, calculated\_coefficients, intercept, and a threshold of 0.5 as parameters. Store the results in final\_results, and print final\_results.

import numpy as np  
from exam import hours\_studied, calculated\_coefficients, intercept  
  
def log\_odds(features, coefficients,intercept):  
 return np.dot(features,coefficients) + intercept  
  
def sigmoid(z):  
 denominator = 1 + np.exp(-z)  
 return 1/denominator  
  
# Create predict\_class() function here  
def predict\_class(features, coefficients, intercept, threshold):  
 calculated\_log\_odds = log\_odds(features, coefficients,intercept)  
 probabilities = sigmoid(calculated\_log\_odds)  
 return np.where(probabilities>=threshold, 1, 0)  
  
# Make final classifications on Codecademy University data here  
final\_results = predict\_class(hours\_studied,calculated\_coefficients,intercept,0.5)  
print(final\_results)

[[0]

[0]

[0]

[0]

[0]

[0]

[0]

[0]

[0]

[1]

[1]

[1]

[1]

[1]

[1]

[1]

[1]

[1]

[1]

[1]]

# Scikit-Learn

Now that you know the inner workings of how Logistic Regression works, let's learn how to easily and quickly create Logistic Regression models with sklearn! [sklearn](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) is a Python library that helps build, train, and evaluate Machine Learning models.

To take advantage of sklearn's abilities, we can begin by creating a LogisticRegression object.

model = LogisticRegression()

After creating the object, we need to fit our model on the data. When we fit the model with sklearn it will perform gradient descent, repeatedly updating the coefficients of our model in order to minimize the log-loss. We train — or fit — the model using the .fit() method, which takes two parameters. The first is a matrix of features, and the second is a matrix of class labels.

model.fit(features, labels)

Now that the model is trained, we can access a few useful attributes of the LogisticRegression object.

* model.coef\_ is a vector of the coefficients of each feature
* model.intercept\_ is the intercept b\_0

With our trained model we are able to predict whether new data points belong to the positive class using the .predict() method! .predict() takes a matrix of features as a parameter and returns a vector of labels 1 or 0 for each sample. In making its predictions, sklearn uses a classification threshold of 0.5.

model.predict(features)

If we are more interested in the predicted probability of the data samples belonging to the positive class than the actual class, we can use the .predict\_proba() method. predict\_proba() also takes a matrix of features as a parameter and returns a vector of probabilities, ranging from 0 to 1, for each sample.

model.predict\_proba(features)

Before proceeding, one important note is that sklearn's Logistic Regression implementation requires feature data to be normalized. [Normalization](https://www.codecademy.com/paths/data-science/tracks/dspath-supervised/modules/dspath-classification/articles/normalization) scales all feature data to vary over the same range. sklearn's Logistic Regression requires normalized feature data due to a technique called Regularization that it uses under the hood. Regularization is out of the scope of this lesson, but in order to ensure the best results from our model, we will be using a normalized version of the data from our Codecademy University example.

**1.**

Let's build, train and evaluate a Logistic Regression model in sklearn for our Codecademy University data! We've imported sklearn and the LogisiticRegression classifier for you. Create a Logistic Regression model named model.

**2.**

Train the model using hours\_studied\_scaled as the training features and passed\_exam as the training labels.

**3.**

Save the coefficients of the model to the variable calculated\_coefficients, and the intercept of the model to intercept. Print calculated\_coefficients and intercept.

**4.**

The next semester a group of students in the Introductory Machine Learning course want to predict their final exam scores based on how much they intended to study for the exam. The number of hours each student thinks they will study, normalized, is given in guessed\_hours\_scaled. Use model to predict the probability that each student will pass the final exam, and save the probabilities to passed\_predictions.

**5.**

That same semester, the Data Science department decides to update the final exam passage model to consider two features instead of just one. During the final exam, students were asked to estimate how much time they spent studying, as well as how many previous math courses they have taken. The student responses, along with their exam results, were split into training and test sets. The training features, normalized, are given to you in exam\_features\_scaled\_train, and the students' results on the final are given in passed\_exam\_2\_train.

Create a new Logistic Regression model named model\_2 and train it on exam\_features\_scaled\_train and passed\_exam\_2\_train.

**6.**

Use the model you just trained to predict whether each student in the test set, exam\_features\_scaled\_test, will pass the exam and save the predictions to passed\_predictions\_2. Print passed\_predictions\_2.

Compare the predictions to the actual student performance on the exam in the test set. How well did your model do?