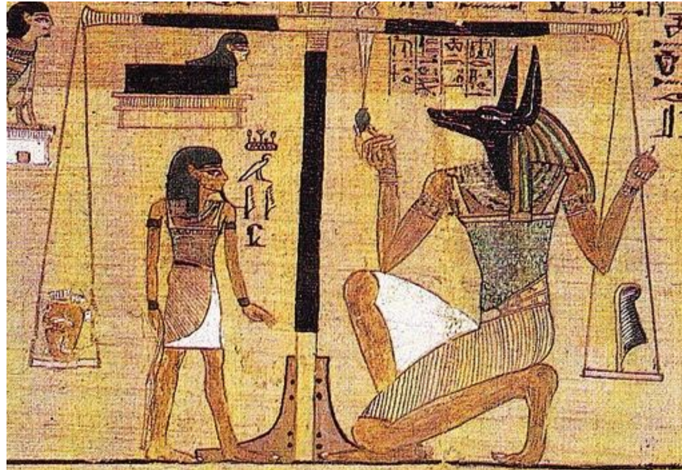
Logistic Regression
ML_2022: Machine Learning

https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/logistic_lecture/logistic_lecture.pdf



Egyptian god Maat weighed the sins of your heart against a feather.

The Logistic Regression Problem

Logistic regression handles simple decision making.

- *We often need to make a Yes or No choice using imperfect knowledge;*
- *Perhaps the only evidence we have is a record of our previous decisions;*
- *We seek a formula that can "explain" our previous decisions;*
- *This formula can be used to predict the right choice in new cases;*
- *We would also like to estimate how likely our decision is to be correct.*

As a classic example of decision making under uncertain knowledge, consider almost any medical diagnosis. The human body is a complicated and varied object, and few mathematical formulas are available to tell us exactly how to deal with a given case. After collecting a number of measurements, a doctor must often decide Yes or No, perform a certain procedure or not. In the absence of a real scientific theory, this means the decision, even though it is based on the evidence of the measurements, is ultimately made simply by intuition built up over years of experience. So what's a new doctor supposed to do?

Suppose that a doctor must decide whether a pregnant woman should have her baby by Caesarian section. A number of medical measurements would seem to be relevant before making the decision, but ultimately, the doctor has to make the choice. If the patient expresses concerns about this decision, the doctor would like to be able to explain whether the decision is strongly recommended or not.

This is an example of a **classification problem**. Our task is to look at a particular case, and decide which class it belongs to. In the *logistic regression* problem, there are only 2 such classes, which might be Yes/No, 0/1, Admit/Reject, Pass/Fail.

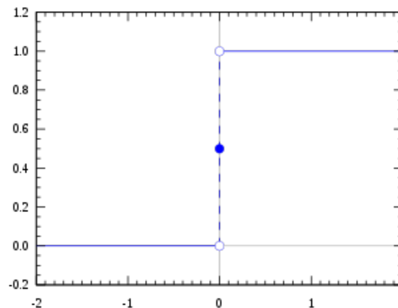
We will try to construct a formula that will produce a good decision based on our past experience. To do so, our first tool will be a mathematical function that does a reasonable job of representing a Yes/No function as a sequence of values that smoothly transition from 0 (No!) to 1 (Yes!).

1 The logistic function

To begin with, let us suppose that the output decision y is to made based on the value of just a single input x. Our decision will be simplified if we can somehow come up with a function $y(x)$ that automatically returns a reasonable Yes or No value. The simplest such function might simply apply some cutoff value x^*:

$$y(x) = \begin{cases} \text{No} = 0 & \text{if } x \leq x^* \\ \text{Yes} = 1 & \text{if } x^* < x \end{cases}$$

The Heaviside function has this behavior:



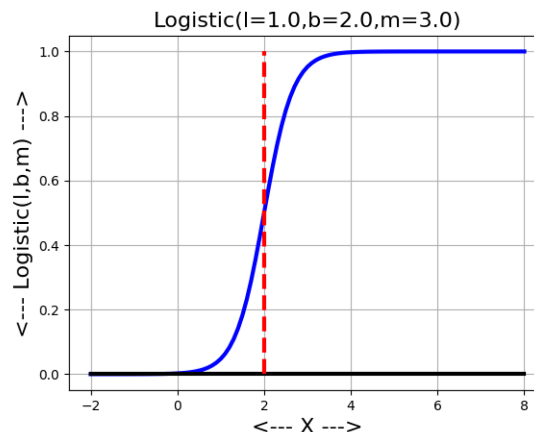
The Heaviside function.

However, this approach does not give us the additional information of how sure we are of our classification, and it does not generalize easily to situations involving multiple input variables x.

Consider the formula known as the *logistic* or *sigmoid function*:

$$y(x) = \frac{l}{1 + e^{-m(x-b)}}$$

which has parameters l (maximum value), m (slope) and b (cutoff). For our purposes, we will always set $l = 1$.



The logistic curve for $l = 1, b = 2, m = 3$.

The function $y(x)$ is near 0 for values to the left of the cutoff, hits $\frac{1}{2}$ at the cutoff value, and then rises to 1 on the right. The location of the cutoff is controlled by the value of b. The sharpness of the rise, and the width of the "uncertainty" region, depend on the slope m.

For a given problem, if we can determine good values of b and m, then we will have a way of classifying our data, as well as reporting how sure we are of our decision:

$$y(x) = \begin{cases} \text{No! (90\% sure)} & \text{if } y(x) <= 0.10 \\ \text{No} & \text{if } 0.10 < y(x) < \frac{1}{2} \\ 50/50 & \text{if } y(x) = \frac{1}{2} \\ \text{Yes} & \text{if } \frac{1}{2} < y(x) < 0.90 \\ \text{Yes! (90\% sure)} & \text{if } 0.90 < y(x) \end{cases}$$

We could tabulate the value of the logistic function with parameters $l = 1, b = 2, m = 3$, over the range $-2.0 \leq x \leq 8.0$.

```
 x     y=logistic(1,2,3,x)

-2     0.000006
-1     0.000123
 0     0.002472
 1     0.047425
 2     0.500000
 3     0.952574
 4     0.997527
 5     0.999877
 6     0.999994
 7     1.000000
 8     1.000000
```

Listing 1: Sample logistic function values.

The table verifies that the logistic function starts out essentially 0 and climbs to its maximum value `l` (which here is 1), and at the cutoff point `b` (which here is 2) it reaches its halfway value `l/2` (which here is 1/2). The role of the slope `m` is harder to see except in a plot. It controls how rapidly the function jumps from its low to high value.

For our purposes, `l` will always be 1, and, assuming we are working with a scalar variable `x`, we will rewrite the formula in terms of two `weights` stored in the vector `w`:

$$y(x) = \frac{1}{1 + e^{(-w_0 - w_1 x)}}$$

We will now investigate how to determine the values `w` that help us to systematize our decision making.

2 *logistic_regression()*

To construct this formula, we will work with a version of the **logistic function**. As we have done in the past, we will modify our original data items `x` by inserting an initial value $x_0 = 1$. In that case, our formula becomes

$$y(x) == \frac{1}{1 + e^{(-w_0 x_0 - w_1 x_1)}} = \frac{1}{1 + e^{-w' x}}$$

The missing information here is the value for the weight vector w.

Suppose we want to set up our formula based on a set of n data items X and 0/1 classification vector y. We can determine the values w if we assume we want to minimize the mean square error between the actual

and computed class values for the data items, that is:

$$\text{minimize } mse(w) = \frac{1}{n} \sum_{i=0}^{i<n} (y_i - y(x_i))^2$$

We have solved this kind of problem before, using the method of gradient descent. We can construct a similar procedure for problems that have been posed with the logistic function.

Consider the following function: `logistic_regression(X,y,alpha,kmax)`, where

- `X` is our normalized input data, a $n \times d$ array with an initial column of 1's
- `y` is our output data, a vector of values of 0 or 1, of length `n`
- `alpha` is a learning rate, which might be 0.1 or 0.01
- `kmax` is the maximum number of iterations

The function has the form:

```
def logistic_regression ( X, y, alpha, kmax ):

  import numpy as np

  n, d = X.shape
  w = np.zeros ( d )

  for k in range ( 0, kmax ):

    y2 = 1.0 / ( 1.0 + np.exp ( - np.matmul ( X, w ) ) )

    for j in range ( 0, d ):
      w[j] = w[j] - ( alpha / n ) * np.dot ( ( y2 - y ), X[:,j] )

  return w
```

While we probably have to experiment to determine satisfactory values of `alpha` and `kmax`, this function has the potential to compute the necessary weight values for our logistic formula.

Now we need a sample data set in order to try this idea out.

3 Apples and Oranges

A fruit company receives truckloads of apples and oranges mixed together. A robot is to be used to sort them. The robot can determine the weight x of each item, and then must compute $y(x)$, which is 0 if the item should go in the orange box, and 1 if it should go in the apple box. This is an example of "logistic classification". We have two categories or classes, and we need to assign each data item x to one of those classes.

In order for the robot to do its job, we need to construct a formula for $y(x)$. Instead of simply returning 0 or 1, our formula will return a number between 0 and 1, representing how strongly it thinks a given fruit is an orange or apple. All values below $\frac{1}{2}$ would be regarded as oranges, while those above the limit are assumed to be apples.

Here is how it would be used for our apples and oranges problem:

```
  import numpy as np

  data = np.loadtxt ( 'apple_data.txt' )

  g = data[:,0]
```

```
y = data[:,1]
n = len ( g )

gmin = np.min(g)
gmax = np.max(g)
gn = ( g - gmin ) / ( gmax - gmin )

X = np.zeros ( [ n, 2 ] )
X[:,0] = 1
X[:,1] = gn

alpha = 0.02
kmax = 100000

from logistic_regression import logistic regression

wn = logistic_regression ( X, y, alpha, kmax )
```

Listing 2: Linear regression for gold coin data.

The weights $nw = [-8.99, 17.88]$ are returned for the normalized data values `gn`. Since it is more natural for us to work with data in the original units, we need to work backwards now.

```
y = 1.0 / ( 1.0 + np.exp ( - wn[0] - wn[1] gn[1] ) )

- wn[0] - wn[1] gn[1] = - wn[0] - wn[1] * ( g - gmin ) / ( gmax - gmin )
                      = - wn[0] + wn[1] * gmin / ( gmax - gmin )
                               - wn[1] / ( gmax - gmin ) * g

w[0] = wn[0] - wn[1] * gmin / ( gmax - gmin )
w[1] =         wn[1] / ( gmax - gmin )
```

So now we know how to convert the weights so that they can be used with our original data.

```
w = np.array ( [ wn[0] - wn[1] * gmin / ( gmax - gmin ), wn[1] / ( gmax - gmin ) ] )
print ( '' )
print ( '  Weights for original data W = (%g,%g)' % ( w[0], w[1] ) )
cutoff = - w[0] / w[1]
print ( '  Cutoff value is ', cutoff )
```

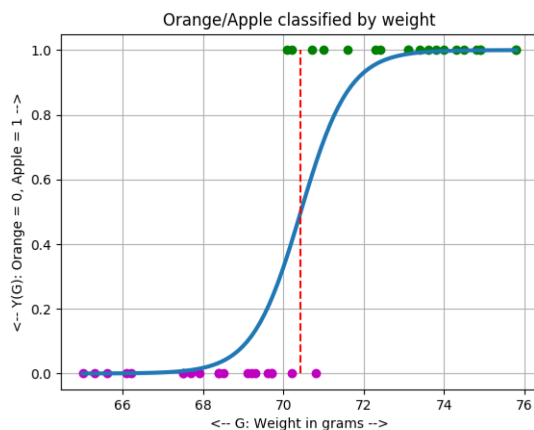
So now we can plot our data and our formula using the original units. The converted coefficients are `w = (-116.606,1.65557)`, and the cutoff value is `- w[0]/w[1] = 70.432`, which tells us the weight at which we can separate the fruit.

To make a nice plot of the results, we want to show our original data, split into the two fruit kinds, and our logistic regression function which tries to summarize the situation.

```
gplot = np.linspace ( gmin, gmax, 101 )
yplot = 1.0 / ( 1.0 + np.exp ( - ( w[0] + w[1] * gplot ) ) )

plt.plot ( g[y==0], y[y==0], 'mo' )                     # oranges in magenta
plt.plot ( g[y==1], y[y==1], 'go' )                     # apples in green
plt.plot ( gplot, yplot, 'b-', linewidth = 3 )          # logistic function in blue
plt.plot ( [cutoff,cutoff], [0,1], '--', color = 'r' )  # cutoff in red
plt.show ( )
```

Our plot suggests how the function $y(x)$ tries to sort the apples and oranges by weight:



The logistic regression curve for the apples and oranges weight data.

4 *LogisticRegression* from Scikit-Learn

We'd like to avoid having to worry about the right settings of `alpha` and `kmax` for the simple `logistic_regression()` function. Hence we will prefer to use a professionally-written implementation of logistic regression, as supplied by Scikit-Learn. However, to do so, we will have to figure out some of the quirks of that code.

We will do so by applying logistic regression to a new dataset, which involves two independent variables. Thus the data will appear as dots scattered across the plane, and logistic regression will try to determine a line that separates the data into the two kinds that have been identified.

Thus, we will have two headaches: dealing with the Scikit-Learn function, and properly handling a dataset involving two variables.

5 Exercise: Admission Statistics

The file *admit_data.txt* contains $n = 100$ records relating to college admission applications. Each record contains a student's scores on an English exam (*eng*), on a math exam, (*mat*), and an admissions decision *adm*, which is 0 (not admitted) or 1 (admitted).

Our task is to find a logistic formula which for student i, is given the data $x_i = [1, eng, mat]$ and returns a y value between 0 and 1, estimating the probability that the student will be admitted. We hope that in general, $y(x_i) \approx adm_i$. If so, we can fire the dean of admissions and just use a robot and this formula instead!

We start by reading the data in the usual way. Because we will be using better software, we won't need to normalize this data in advance!

```
#
#  Read the data: (x1=english, x2=math, adm=admit(0/1))
#
  data = np.loadtxt ( 'admit_data.txt' )
  eng = data[:,0]
  mat = data[:,1]
  adm = data[:,2]
  n = len ( eng )
#
#  Create arrays X and y.
```

```
#
  X = np.zeros ( [ n, 3 ] )
  X[:,0] = 1
  X[:,1] = eng[:]
  X[:,2] = mat[:]

  y = adm[:]
```

Listing 3: Read and prepare admissions data.

Now we need to crank out the formula. First we have to import the `LogisticRegression()` function. Then we feed it our `X,y` data, and it returns an object `classifier` which contains the logistic regression model. By calling `classifier.predict()`, we can see how the model classifies our original data. Since we need them for plotting, we have to retrieve the coefficient array `w`; doing this is a little tricky, involving underscores in variable names, and also getting `w[0]` from a separate location. But once we've done all that, we have the information we need.

```
  from sklearn.linear_model import LogisticRegression
#
#  Fit the logistic regression model to the data.
#
  classifier = LogisticRegression(random_state=0).fit ( X, y )
#
#  Evaluate the model on the original data.
#
  yp = classifier.predict ( X )
#
#  Compute MSE
#
  mse = ( 1.0 / n ) * sum ( ( yp - adm )**2 )
#
#  Construct the coefficient array W.
#
  w0 = classifier.intercept_
  w = classifier.coef_[0]
  w[0] = w0

  print ( '' )
  print ( '  Computed weights W = ', w )
```

We know how to display our data, but drawing the separating line is a little complicated now. But we know that all points `(eng,mat)` on the separating line satisfy the equation

$$-w_0 - w_1 eng - w_2 mat = 0$$

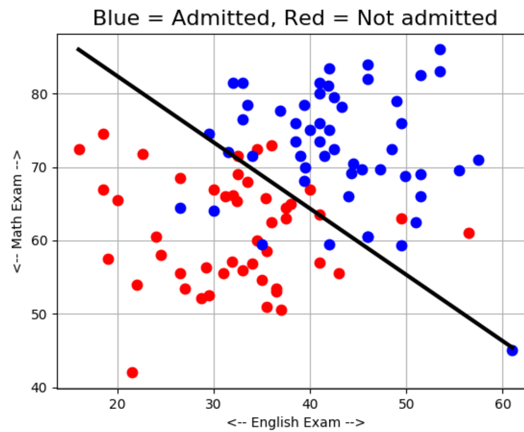
So, if `emin` is the minimum value of `eng`, the plot of the dividing line can start at `emin, (-w[0]-w[1]*emax)/w[2])`, with a corresponding endpoint at `emax`.

```
#
#  Display data.
#
  emin = np.min ( eng )
  emax = np.max ( eng )
  mmin = ( - w[0] - w[1] * emin ) / w[2]
  mmax = ( - w[0] - w[1] * emax ) / w[2]

  plt.plot ( X[y==0,1], X[y==0,2], 'r.', markersize = 15 )
  plt.plot ( X[y==1,1], X[y==1,2], 'b.', markersize = 15 )
  plt.plot ( [emin,emax], [mmin,mmax], 'k-', linewidth = 3 )
  plt.show ( )
```

Listing 4: Display the data

Our plot shows that most students were admitted or not according to a linear formula, although there are clearly some exceptions!



The logistic regression curve for the college admissions data.

Three students apply late. We will use the weights w to decide whether to admit them. The raw data is:

```
eng   mat
 46    81
 26    75
 30    66
```

We create a new `X2` array for this data. If we call `classifier.predict()`, we will get a 0 or 1 result returned, meaning don't admit/admit. If instead we call `classifier.predict_proba()`, we get two values, the probability that the student should not be admitted, and the probability that the student should be admitted.

```
X2 = np.array ( [ \
    [ 1.0, 46, 81 ], \
    [ 1.0, 26, 75 ], \
    [ 1.0, 30, 66 ] ] )
y2 = classifier.predict ( X2 )
p2 = classifier.predict_proba ( X2 )
```

Listing 5: Do we admit these new students?

Here are the results for this data:

```
New student data: (X2)
  [[ 1. 46. 81.]
   [ 1. 26. 75.]
   [ 1. 30. 66.]]

Admit new student? (y2)
  [1. 0. 0.]

Strength of decision (p2)
  [[0.00626659 0.99373341]
   [0.61183432 0.38816568]
   [0.84456637 0.15543363]]
```

So...fire the dean of admissions!

6 Gopher Heads

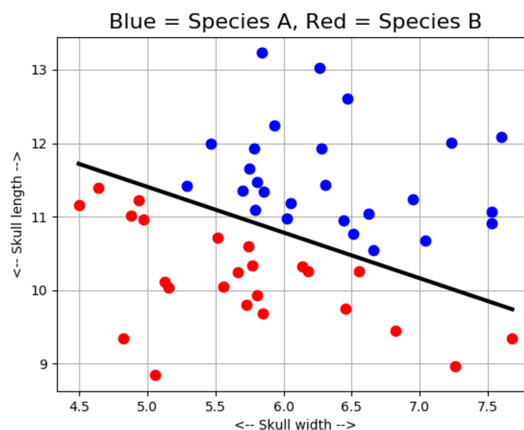
Last summer, our biology project required students to go into the prairie in search of gophers. There are two gopher species there, “Arctic Gophers” (species A) and “Balkan Gophers” (species B). The students measured the width and length of each the skull of each gopher, and the instructor was able to determine the species. Unfortunately, the instructor wrote down a “1” for Arctic gophers, and a “-1” (not 0!) for the Balkan gophers.

This summer, the instructor cannot accompany the students, so they will go back to the prairie and make their measurements, but they need help to estimate the species for each skull. If we are lucky, we will be able to come up with a formula which relates the species to the measurements they have made.

Our data is stored in the file *gopher_data.txt*, and holds $n = 50$ records of the form `swi, sle, spe`, that is, skull length, skull width, and species. We will create the $n \times 3$ array `X` containing `[1,swi,sle]`. To create the result vector `y`, we can't simply copy the values in `spe` since those involve -1 instead of 0. One way to handle this would be to consider a logical expression, and turn the result back into an integer:

```
y = ( spe == 1 )
y = y.astype ( int )
```

Now the computation of the logistic regression classifier proceeds in the same way as for the college admission data, and we can create a plot of our findings:



The logistic regression classification for gopher species.

Suppose the students report the following data from their followup project:

```
swi  sle
---  ---
6.5  10.8
6.0  10.0
5.6   9.3
```

Using `classifier.predict()` we compute [1,0,0]. Using `classifier.predict_proba()` we get the more informative probability results:

```
prob not A    prob A
----------    ----------
0.29986994    0.70013006    <-- somewhat sure
0.88461229    0.11538771    <-- pretty sure
0.98901838    0.01098162    <-- extremely sure!
```

7 Diabetes Prediction

A medical team has compiled records on $n = 768$ female patients who were tested for diabetes. The measurements included:

- `prg`: number of pregnancies;
- `glu`: plasma glucose concentration;
- `dbp`: diastolic blood pressure;
- `tri`: triceps skinfold thickness;
- `ins`: insulin level;
- `bmi`: body mass index;
- `ped`: diabetes pedigree function;
- `age`: age;
- `dia`: nondiabetic(0) / diabetic(1);

Because they can easily make the 8 medical measurements, while the diabetic diagnosis itself is more expensive and time consuming, they would appreciate a formula that allows them to estimate the likelihood that a patient is diabetic.

Because this case involves so many independent variables, we won't be able to make a simply plot that cleary summarizes the results. We also have to consider that having so many factors tends to make it easier to separate the two classes, but having so many cases makes it more difficult. We will have to see how well our formula does.

Unfortunately, when we run our code, we get an error message, whose content involves a warning we have seen before:

```
  Logistic regression for diabetes diagnosis data.

  ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
```

We are familiar with the fact that badly scaled data can make convergence difficult. Rather than normalize the data ourselves, we can actually rely on a built in Scikit-Learn function, as described on the preprocessing web page. Our use is as follows:

```
#
#  Scale the data.
#
  from sklearn import preprocessing
  scaler = preprocessing.StandardScaler().fit ( X )
  X_scaled = scaler.transform ( X )
#
#  Fit the logistic regression model to the data.
#
  classifier = LogisticRegression(random_state=0).fit ( X_scaled , y )
#
#  Evaluate the model on the original data.
#
  yp = classifier.predict ( X_scaled )
```

As you can see, once we have computed `X_scaled`, we have to use these scaled values instead of `X` for our calls to `classifier()`. If we compute the weights `w`, then these also must use the scaled data.

Once we have converted to scaled data, our results come quickly:

```
diabetes():
  Logistic regression for diabetes diagnosis data.
  MSE =  0.21614583333333331

  Computed weights W =  [-0.86678294  0.40864687  1.10711197 -0.25086804  0.00905046 -0.13083627
  0.69630872  0.30883724  0.17649782]
```

If we need to use `classifier()` to predict the results for new data `X_new`, then we can simply apply `X_new_scaled = scaler.transform(X_new)`. If, on the other hand we really really want to work with our unscaled data, then we have seen before how to convert coefficients of normalized data to coefficients for original data.