Computing Model Likelihood

Bookmark this page

To learn our logistic model through optimization:

1. We compute the log-likelihood function, which is the loss function of our model.
2. We compute the derivative of the log-likelihood function for gradient descent optimization.

1) Computing the log-likelihood function

We compute how well our logistic model fits the data by comparing its output probabilities with the labels for each data point, using a likelihood function.

The likelihood function for our i (between 0 and 1) and the real i (either 0 or 1) is given by:

$$\mathcal{L}(p_i \mid Y_i) = P(Y_i = y_i) = p_i^{y_i}(1-p_i)^{(1-y_i)}$$

We use this formula because when the label y_i is 1, the likelihood is p_i , and when the label y_i is 0, the likelihood is 1 - p_i.

To get our total likelihood, we multiply the likelihoods of all our data points (using the capital greek letter Pi to represent a product for all i)

$$\mathcal{L}(p|Y) = \prod_i \mathcal{L}(p_i|Y_i)$$

However, an easier function to optimize is the log-likelihood.

Review from previous section

The log function is "convex", so if we find an optimum for the log-likelihood, it is equivalently an optimum for the total likelihood. And since the log-likelihood is an expression of a sum instead of a product, it is computationally easier to optimize. To turn the procedure into a minimization instead of a maximization, we seek to minimize negative log-likelihood instead of maximize log-likelihood.

Our log-likelihood function is

$$l(p|Y) = -\log(\mathcal{L}(p|Y))$$

We simplified the equation by converting the log of a product into a sum of logs

$$-\log(\mathcal{L}(p|Y)) = -\log(\Pi_i(\mathcal{L}(p_i|Y_i))) = -\Sigma_i(\log(\mathcal{L}(p_i|Y_i)))$$

We simplify the equation again by converting a log of products into a sum of logs

$$-\Sigma_i(\log(\mathcal{L}(p_i|Y_i))) = -\Sigma_i(\log(p_i^{y_i}(1-p_i)^{(1-y_i)})) = -\Sigma_i(\log(p_i^{y_i}) + \log((1-p_i)^{1-y_i}))$$

Finally we move the exponents within the log term to be products outside the log term

$$-\Sigma_i(\log(p_i^{y_i}) + \log((1-p_i)^{1-y_i})) = -\Sigma_i(y_i\log(p_i) + (1-y_i)\log(1-p_i))$$

Our loss function will be the average log-likelihood of our data points instead of the total log-likelihood, so that we can equally compare the log-likelihoods when the number of data points changes.

$$Loss(p|Y) = -\frac{1}{N}\Sigma_i(y_i\log(p_i) + (1-y_i)\log(1-p_i))$$

Here is a NumPy implementation of this loss function which is vectorized, meaning you can use it for any number of beta parameters in a vector called `beta`. Our work so far has assumed that $\beta = [\beta_0, \beta_1]$ for a one-dimensional dataset, X_{train}. But the code below would also work for $\beta = [\beta_0, \ldots, \beta_{10}]$ for a ten-dimensional X_{train}.

```
sigmoid = lambda x : 1 / (1 + np.exp(-x))
def loss(X, y, beta, N):
    # X: inputs, y : labels, beta : model parameters, N : # datapoints
    p_hat = sigmoid(np.dot(X, theta))
    return -(1/N)*sum(y*np.log(p_hat) + (1 - y)*np.log(1 - p_hat))
```

2) Computing the derivative of the log-likelihood function for gradient descent optimization

To optimize our logistic regression model, we want to find beta parameters to minimize the above equation for negative log-likelihood. This corresponds to finding a model whose output probabilities most closely match the data labels.

To compute our derivative, we first calculate the derivative of loss with respect to the probabilities p_i, then calculate the derivative of the probabilities with respect to our beta parameters. Using the chain rule, we will derive a simple formula for the derivative of our loss.

Calculating the derivative of the loss with respect to our model probabilities

Since the derivative of log(x) is 1/x, the derivative of our loss function with respect to the probabilities p_i is

$$\frac{d\,\text{Loss}\,(p|Y)}{d\,p} = -\frac{1}{N}\Sigma_i\,(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i})$$

The derivative of p_i with respect to our Beta parameters (using the chain rule) is

$$\frac{dp_i}{d\beta} = \frac{d}{d\beta}(\frac{1}{1+e^{-(\beta x_i)}}) = \frac{-1}{(1+e^{-(\beta x_i)})^2}(e^{-(\beta x_i)})\,(-x_i) = \frac{1}{(1+e^{-(\beta x_i)})^2}(e^{-(\beta x_i)})\,x_i$$

To simplify this equation, we can re-write it directly in terms of the probabilities p_i:

$$= \frac{1}{(1+e^{-(\beta x_i)})}\frac{e^{-(\beta x_i)}}{(1+e^{-(\beta x_i)})}x_i = p_i\,(1-p_i)\,x_i$$

Calculating the derivative of the loss with respect to our model parameters

Due to the chain rule, the derivative of our loss is

$$\frac{d\,\text{Loss}\,(p|Y)}{d\beta} = \frac{d\,\text{Loss}\,(p|Y)}{dp}\frac{dp}{d\beta} = \Sigma_i\,[\frac{d\,\text{Loss}\,(p_i|Y_i)}{d\,p_i}\frac{dp_i}{d\beta}]$$

$$= -\frac{1}{N}\Sigma_i\,[(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i})\,p_i\,(1-p_i)\,x_i]$$

$$= -\frac{1}{N}\Sigma_i\,[(y_i\,(1-p_i) - (1-y_i)\,p_i)\,x_i]$$

This final sum of products can be re-written as a dot product in the following way, making our Python implementation simpler:

$$= -\frac{1}{N}[Y\,(1-p) - (1-Y)\,p]\cdot X$$

Here is a NumPy implementation of the derivative of our loss function.

```
def d_loss(X, y, theta, N):
    # X: inputs, y : labels, beta : model parameters, N : # datapoints
    p_hat = sigmoid(np.dot(X, beta))
    return np.dot(X.T,-(1/N)*(y*(1 - p_hat) - (1 - y)*p_hat))
```

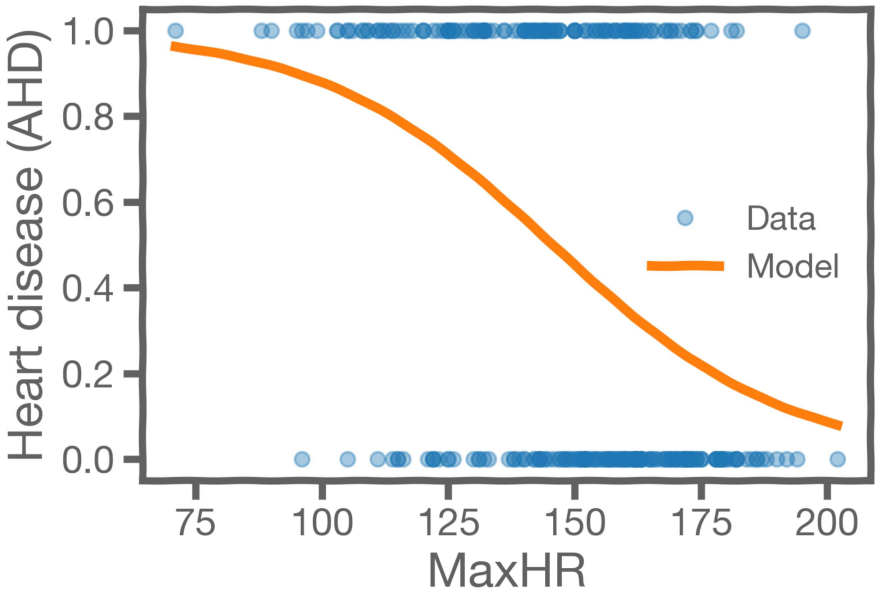
To use gradient descent, we start with an initial guess for our Beta parameters, and then update them by small steps in the direction that results in fitting the data better. That is, in the direction of lowest negative log-likelihood. After each step we can compute the loss to track the progress of the optimization.

```
step_size = .5
n_iter = 500
beta = np.zeros(2) # initial guess for Beta_0 and Beta_1
losses = []
for _ in range(n_iter):
    beta = beta - step_size * d_loss(beta)
    losses.append(loss(beta))
```

Turning probabilities into discrete classifications

Turning probabilities into discrete classifications

The model below can act as a classifier for the heart disease dataset which contains a lot of overlap. The height of the orange model curve represents our model's probability that a patient with that maxHR has heart disease. Notice that for a given patient's maximum heart rate close to the mean of our dataset, like 130, it is really unclear whether the patient is in fact diagnosable with heart disease. In this case, our model predicts close to 50% probability of heart disease. On the other hand, if the max heart rate is 190, our model would predict close to 10% probability of heart disease, or if the max heart rate is 85, our model would predict close to 90% probability of heart disease.



Turning model probabilities into concrete classifications— for example, declaring that a patient is a YES, NO, or MAYBE on whether or not they have heart disease— requires choosing a classification thresholds. A common threshold is 50%, (predict YES for >50% and NO for < 50%) but for certain applications it may make sense to choose a different threshold.

Let p be the probability output by our model. Because our model here contains significant overlap, it may be sensible to classify a patient as YES when $p \geq 90$, classify a patient as NO when $p \leq 10$, and classify a patient as MAYBE when $p = 50$.

Discussion Board (External resource)

Click OK to have your username and e-mail address sent to a 3rd party application.

OK

‹ Previous | Next ›

Connect

Calculator