

Logistic Regression

☰ Week	TUES. Week 5
☰ Assignment Due	
☑ Assignment Done	<input type="checkbox"/>
📅 Due Date	
☑ Notes Done	<input checked="" type="checkbox"/>

Presentation

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4efe74ee-3dae-4283-a9a5-dad7d4cc8596/L1-L2-LogisticReg.pdf>

Class Notes

Logistic Regression

Currently, we've talked about LDA, KNN (for binary or multi-class classification), statistical classification (for binary or multi-class classification), and decision trees.

- The next algorithms we'll talk about, **logistic regression** and **support vector machines (SVMs)** are mathematically framed for *binary* classification.
 - **Logistic regression** is one of the building blocks for artificial neural networks!
- **Logistic regression** is actually a classification algorithm, despite its name!
 - Name comes from it being extremely similar to a common type of regression called **linear regression**, which we'll discuss next week.
 - It was created specifically for binary classification problem and it returns a value between 0 and 1 (a valid probability).
 - The probability returned is the probability of one of the classes (often referred to as the *positive class*, with $y = 1$):
 - $0 \leq P(y = 1|\mathbf{x}) \leq 1$
 - The way we compute this value $P(y = 1|\mathbf{x})$ is according to the function:

$$P(y = 1|\mathbf{x}) = g(\mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{x}\mathbf{w} + b)}}$$

- Where:
 - \mathbf{w} is a (column) vector of weights, one per feature of x
 - b is a *bias* weight → think of as an “offset”
- The function $g(z) = \frac{1}{1+e^{-z}}$ is referred to as the **sigmoid** or **logistic function** (or **logistic sigmoid**)
 - The value z is computed by taking our observation, multiplying by some weight and then adding in some bias value.
 - The learning that comes in is through finding out what should the weights be in the vector \mathbf{w} and what the bias b should be s.t. when we do this computation, we get optimal results.
 - This function tends to 0 as z decreases and tends to 1 as z increases (positive association).
 - For some value z that is unbounded, the function will map it to a value within the range 0 and 1, thus why it can be used as a valid probability value.
 - In addition to resulting in values that can be interpreted as probabilities, it has a nice characteristic in that it's differentiable.

Fit Parameters Based on Maximum Likelihood

Once we know the weights \mathbf{w} and bias b , we can plug everything into the equation, which gives us a value that is interpreted as a probability of class 1 given an observation.

- Since we're considering a classification problem, the probability of class 0 is $P(y = 0 | \mathbf{x}) = 1 - g(\mathbf{x}) = 1 - \frac{1}{1+e^{(\mathbf{xw}+b)}}$
 - Key: Figure out parameters \mathbf{w} and b s.t. we minimize the classification error or conversely to find the parameters that maximize the correct class likelihood.
- Given a supervised observation (x, y) , we'll let \hat{y} be our prediction that $y = 1$
 - Which again, for logistic regression is computer as $\hat{y} = \frac{1}{1+e^{(\mathbf{xw}+b)}}$
- Initially, an evaluation function would look something like this:
 - $J = \begin{cases} \hat{y} & y = 1 \\ 1 - \hat{y} & y = 0 \end{cases}$
- We can use the function J to tell us how well we did, which we can call the **likelihood** that we are correct, which we can calculate as:

$$J = \ell(y|\mathbf{x}) = (\hat{y})^y (1 - \hat{y})^{(1-y)}$$

- If $y = 1$, the second term would go to 0, and we'd have \hat{y} telling us how we did (how confident we are in our calculations).
- We want to **MAXIMIZE** the likelihood J .
- The process of finding the weights to maximize the likelihood is called the **maximum likelihood estimate (MLE)**.

Log Likelihood

To find the optimal weights, we'll use calculus, since we know our logistic sigmoid function is differentiable!

- Instead of only trying to maximize this J value, we typically try to maximize the **logarithm** of the likelihood to make the calculus a bit nicer for us.

From the properties of logarithms

- $\log_b(mn) = \log_b(m) + \log_b(n)$
- $\log_b(m^n) = n \cdot \log_b(m)$
- In applying logs to our likelihood J , we get:

$$J = y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y})$$

- Now, we refer to this value of the log likelihood.

Log Loss

It is more common to minimize the negation of the log likelihood, which is referred to as the log loss.

$$J = -(y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}))$$

- Typically, we think about trying to *minimize loss*.
 - This turns the *maximization* problem into *minimization* problem

There are typically two strategies taken to solve maximization/minimization problems:

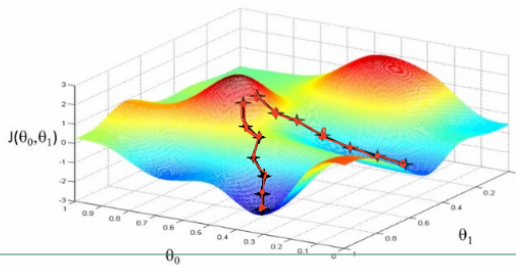
- Strategy #1: Direct approach - Take derivative, set equal to 0, solve for variable
 - Saw in PCA and LDA that we had to use eigen-decomposition to help get solutions
 - Pros: Global solution
 - Cons: May not be computationally feasible or possible via calculus
- Strategy #2: Iterative approach
 - Begin with a guess at weights and iteratively changes the weights s.t. we improve our result
 - We take derivative with respect with our weights and use them to update the weights rather than setting them equal to 0 to solve for them.
 - Pros: Flexible → used in all sorts of optimization problems
 - Cons: May take a while to get to a good solution that may not be the global one! May only reach a locally optimal solution
- For log loss function, there is no direct solution unfortunately 😞

- As a result, we're effectively forced to use an iterative approach.
- NOTE: When doing this approach, it can be greatly advantageous to z-score your data using the training data. It can result in faster convergence.
 - **THIS DEPENDS GREATLY ON THE ALGORITHM THAT YOU ARE USING WHETHER Z-SCORING IS BENEFICIAL.**

Gradient Ascent/Descent

The iterative approach to finding an optimal solution is often called **gradient ascent** or **hill climbing**:

- Note: If what we're trying to do is maximize something, it's gradient **ascent**. If what we're trying to do is minimize something, then it's gradient **descent**.



The graphic depicts gradient ascent with two different initial values of (θ_0, θ_1) and updating each parameter simultaneously

- It should be noted that each of these points start a different points, but end up at the same peak.
- Back to our objective function that we want to minimize → **log loss function**

$$J = -(y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}))$$

- We have a \mathbf{w} vector and b bias for which we need the gradient for each weight:
 - For each weight in \mathbf{w} , w_j , we need $\frac{\partial J}{\partial w_j}$
 - For our bias weight b , we need $\frac{\partial J}{\partial b}$

Derivation of Log-Loss Function Gradient

In trying to take the gradient $\frac{\partial J}{\partial w_j}$, we can begin by noting that our objective function includes logs, where we can recall how to take the derivative of a log:

$$\frac{\partial}{\partial x} (\ln x) = \frac{1}{x} \cdot \frac{\partial}{\partial x} (x)$$

$$\frac{\partial J}{\partial w_j} = - \left(\frac{y}{\hat{y}} \frac{\partial}{\partial w_j} (\hat{y}) + \frac{1-y}{1-\hat{y}} \frac{\partial}{\partial w_j} (1-\hat{y}) \right)$$

- From here, we can factor out the derivative term for $\frac{\partial}{\partial w_j}$.

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \left(-y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}) \right)$$

At this point, we must calculate what the derivative is for \hat{y} with respect to w_j to continue this derivative $\rightarrow \frac{\partial \hat{y}}{\partial w_j}$

must now calc. this to continue the derivation

$$\frac{\partial \hat{y}}{\partial w_j} = \frac{\partial}{\partial w_j} (g(x)) = \frac{\partial}{\partial w_j} \left(\frac{1}{1 + e^{-(xw+b)}} \right) = \frac{\partial}{\partial w_j} (1 + e^{-(xw+b)})^{-1}$$

rewrite \hat{y} rewrite w/ exponents

$$= -1 (1 + e^{-(xw+b)})^{-2} (0 - x_j e^{-(xw+b)}) = \frac{x_j e^{-(xw+b)}}{(1 + e^{-(xw+b)})^2}$$

chain rule j^{th} feat of our observ.

$$= \frac{x_j e^{-(xw+b)}}{(1 + e^{-(xw+b)})^2} = x_j \hat{y} \frac{e^{-(xw+b)}}{1 + e^{-(xw+b)}} = x_j \hat{y} (1 - \hat{y})$$

only term we care abt so we pull

$x_1 w_1 + x_2 w_2 + \dots + x_j w_j$

x = column vector of weights
one weight ea. feat.
 b = bias weight

From earlier:

plug in calc'd derivative

w/ algebraic manipulations

needed for prev. deriv. w/ resp to w_j

Then we can plug in the gradient $\frac{\partial \hat{y}}{\partial w_j}$ in our previous derivative $\frac{\partial J}{\partial w_j}$ that we were trying to calculate:

Column vector of weights
 one weight ea. feat.
 as weight

Caveat: so we pull

$$= \frac{1 \cdot x_j e^{-(xw+b)}}{(1 + e^{-(xw+b)})^2} = x_j \hat{y} \frac{e^{-(xw+b)}}{1 + e^{-(xw+b)}} = x_j \hat{y} (1 - \hat{y})$$

(1 - y) w/ algebraic manipulations

needed for prev. derivative w/ resp to w_j

From earlier:

$$\frac{\partial J}{\partial w_j} = -\left(\frac{y}{\hat{y}} \frac{\partial}{\partial w_j} (\hat{y}) + \frac{1-y}{1-\hat{y}} \frac{\partial}{\partial w_j} (1-\hat{y}) \right)$$

plug in calc'd derivative

simplify

$$= -(y - \hat{y}) x_j = (\hat{y} - y) x_j$$

$w^T x + b$
 \downarrow
 $1 + e^{-(xw+b)}$

The derivation for the $\frac{\partial J}{\partial b}$ would effectively be the same, but at a certain point in the derivation, the x_j term would be 1 from continuing.

$$\frac{\partial J}{\partial w_j} = (\hat{y} - y) x_j$$

$$\frac{\partial J}{\partial b} = (\hat{y} - y) 1$$

$$\frac{\partial}{\partial b} (1 + e^{-(xw+b)})^{-1} = -1(1 + e^{-(xw+b)})^{-2} (0 - 1e^{-(xw+b)})$$

\downarrow w/ further deriv,
 x_j term would be 1
 from continuing

Question: What do we do with these gradients?

- Use them to update our params!
 - Since we're looking to *minimize* the log loss, we'll move some amount in the direction of the **negative** gradient.

$$w_j = w_j + \eta \left(-\frac{\partial J}{\partial w_j} \right)$$

$$b = b + \eta \left(-\frac{\partial J}{\partial b} \right)$$

- Greek letter eta, η , is a **hyperparameter (value we choose)** called the **learning rate**, which controls how much we move in the direction of the gradient.

Gradient Descent Pseudocode

$$\frac{\partial J}{\partial w_j} = (\hat{y} - y)x_j, \quad \frac{\partial J}{\partial b} = (\hat{y} - y)$$

1. We initialize our parameters → typically to some small random numbers between negative and positive 10^{-4}
2. For each update iteration → called an **epoch**
 - a. Grab an observation (x, y) and compute all the gradients using it.
 - b. Update the parameters using their gradients.

Batch Gradient Learning

This above process is **ONLINE gradient learning** because we update the weights using one observation at a time.

- Issue: Using a single observation at a time can take along time to converge.
 - BETTER: Instead of allowing one observation to dictate where we go, we can compute the *average* of the gradients over some set of observations and use this to update the weights.
 - This is referred to as **batch gradient learning**.

Batch Gradient Descent Pseudocode

$$\frac{\partial J}{\partial w_j} = (\hat{y} - y)x_j, \quad \frac{\partial J}{\partial b} = (\hat{y} - y)$$

This will typically be the approach that we'll be implementing!

1. We initialize our parameters → typically to some small random numbers
2. For each update iteration → called an **epoch**
 - a. Set the gradient accumulators to zero: $\frac{\partial J}{\partial b} = 0, \frac{\partial J}{\partial w_j} = 0$
 - b. For each observation you want to use:
 - i. Compute all the gradients for this observation and add them to their associated accumulators.
 - c. Update the parameter according to the average of its gradients (basically divide its accumulator by how many observations contributed to it).

Leveraging Linear Algebra

This approach can take a significant number of loops since we're going through each observation and going through each weight, almost giving us a triply loop.

- Fortunately, we can leverage linear algebra to do these loops efficiently!
 - We don't need to loop overall weights of our weight vector → instead, we can compute all the weights at once using the formula: $\frac{\partial J}{\partial \mathbf{w}} = \mathbf{x}^T(\hat{\mathbf{y}} - \mathbf{y})$
- Further, by going over all observations and *then averaging them*, we can leverage LA again to write the summation overall observations in a crafty linear algebra way!

$$J = \frac{1}{N} \sum_{i=1}^N -(Y_i \ln(\hat{Y}_i) + (1 - Y_i) \ln(1 - \hat{Y}_i))$$

$$= -\frac{1}{N} (Y^T \ln(\hat{Y}) + (1 - Y)^T \ln(1 - \hat{Y}))$$

These matrix multiplications give us the summations that we need

- Working through the linear algebra and calculus, we'd arrive at:

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{N} X^T (\hat{Y} - Y)$$

N - number of observations

The parameter b is just a scalar, so we can just average it over multiple observations.

- Or multiply $\hat{Y} - Y$ by a row vector of ones (to do the summation)

$$\frac{\partial J}{\partial b} = \frac{1}{N} \text{ones}(1, N) (\hat{Y} - Y)$$

This is the equivalent of taking the average of the \hat{y} - y values, which is a column vector.

Batch Gradient Descent Pseudocode with Linear Algebra

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{N} X^T (\hat{Y} - Y), \quad \frac{\partial J}{\partial b} = \frac{1}{N} \text{ones}(1, N) (\hat{Y} - Y)$$

1. Init your params, typically to some small random numbers
2. For each update iteration (also known as **epoch**)
 - a. Compute $\frac{\partial J}{\partial b}$, $\frac{\partial J}{\partial \mathbf{w}}$ as shown above.
 - b. Update the bias and weights by subtracting some amount of these from them.
 - i. $w = w - \eta \frac{\partial J}{\partial w}, b = b - \eta \frac{\partial J}{\partial b}$

Mini-Batching

A disadvantage of using batch gradient descent is that by using all the data at once here, we're slightly more prone to overfitting.

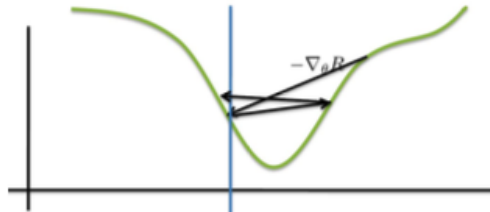
- To strike a balance with online and full batch, we can do **mini batches**:
 - Takes full data, separating them into batches/subsets (called **mini-batches**), which update the parameters with each mini-batch.
 - Once all subsets update the parameters, that is considered the end of the **epoch**.
 - If we shuffle the data, then this mini-batching is referred to as **stochastic mini-batching**

Design Decisions with Mini-Batching

We have several decisions to make:

- What should our batch size be?
 - Smaller batches result in less overfitting (more randomness).

- Larger batches result in quick convergence.
- What should our learning rate be?
 - Smaller learning rate results in slow convergence.
 - Larger learning rate might result in overjumping our extrema.
 - In learning to go to one direction, we may jump way off where we would've found a solution



- What should our termination criteria be?
 - Since we're using an iterative approach, we must ask ourselves how we know when we're "done"?
 - Some ideas to consider:
 - Max number of iterations reached
 - Parameters change very little
 - Change in the objective function is very little.
 - With iterative algorithms, prof typically likes to have 2 stopping criteria:
 - Parameters that show whether we've arrived at convergence
 - **Worst-case scenario** → some amount of time that we're willing to run the algorithm for, max number of iterations

Example

Obs. No.	Y		X-Variables						
	Buy	Income	Is Female	Is Married	Has College	Is Professional	(Omitted Variables)	Prev Child Mag	Prev Parent Mag
1	0	24000	1	0	1	1	...	0	0
2	1	75000	1	1	1	1	...	1	0
3	0	46000	1	1	0	0	...	0	0
4	1	70000	0	1	0	1	...	1	0
5	0	43000	1	0	0	0	...	0	1
6	0	24000	1	1	0	0	...	0	0
7	0	26000	1	1	1	0	...	0	0
8	0	38000	1	1	0	0	...	0	0
9	0	39000	1	0	1	1	...	0	0
10	0	49000	0	1	0	0	...	0	0
...
654	0	10000	1	0	0	0	...	0	0
655	1	75000	0	1	0	1	...	0	0
656	0	72000	0	0	1	0	...	0	0
657	0	33000	0	0	0	0	...	0	0
658	0	58000	0	1	1	1	...	0	0
659	1	49000	1	1	0	0	...	0	0
660	0	27000	1	1	0	0	...	0	0
661	0	4000	1	0	0	0	...	0	0
662	0	40000	1	0	1	1	...	0	0
663	0	75000	1	1	1	0	...	0	0
664	0	27000	1	0	0	0	...	0	0
665	0	22000	0	0	0	1	...	0	0
666	0	8000	1	1	0	0	...	0	0
667	1	75000	1	1	1	0	...	0	0
668	0	21000	0	1	0	0	...	0	0
669	0	27000	1	0	0	0	...	0	0
670	0	3000	1	0	0	0	...	0	0
671	1	75000	1	1	0	1	...	0	0
672	1	51000	1	1	0	1	...	0	0
673	0	11000	0	1	0	0	...	0	0

iversity

KidCreative.csv

- To begin, we shuffle our data → grabbing $\frac{2}{3}$ for training and $\frac{1}{3}$ for validation
- Common starting point is to generate random values within the same 10^{-4} , very small numbers
 - We **z-score** our training data.
 - Noticed that income feature was at much larger scale than others, don't want income feature to artificially larger contribution
- Since our equation is based on log loss, let's terminate when the change in mean of log loss doesn't change more than 2^{-32} or more than 1,000,000 epochs have run

Recall the log loss of an example being correct is

$$-(y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}))$$

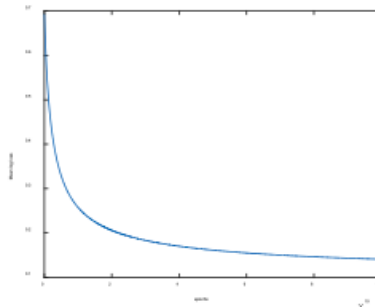
But be careful... $\log(0) = -\text{Inf}$. So you might need to deal with this somehow

- A common thing to do is to add a small *numeric stability constant* within your logs.

- Must be careful with taking the log of 0 since this is negative infinity → **add in a numeric stability constant** 10^{-7}

- Choosing Class 1 if $P(y = 1|x) \geq 0.5$ we get:

	Precision	Recall	F-Measure
Training Set	0.82	0.84	0.83
Validation Set	0.78	0.82	0.80



$w =$
 2.6437
 0.3770
 0.2495
 0.1182
 -0.1997
 -0.2350
 -0.0639
 0.0909
 0.0457
 0.3581
 0.1723
 0.2007
 0.4907
 0.5288
 0.1437
 0.1545
 $b = -3.6519$

A good visualization entails having a good amount of learning happening at the beginning with learning eventually flatline.

- If we think of the features from the dataset and the weights of the features, we can see that salary contributes the most to whether someone may purchase something.
- Upon having training terminate, we can take our model, getting our \hat{y} from our sigmoid function to then calculate the possible classification of a new validation observation.
- Once we have class labels, we can compute precision, recall, f-measure, etc.

When looking at our performance between training set and validation set, we can see that performance was better in the training set.

- There isn't too much of a difference in performance though, telling us that the model did generalize its learning to an extent.

Dealing with Overfitting

Possibly in looking at our dataset, we may see the implication of overfitting.

- Since logistic regression is similar to the gradient based approach of linear regression, it's techniques for dealing with overfitting are similar.
- Generally, we want to:
 - Try to get more data for training → more data = better, full representation of data that training data
 - Could try implementing cross-validation if it is difficult to collect *new* observations
 - Select the training subset better from overall data
 - Perhaps the subset grabbed from training wasn't a fair representation of all data

- Don't use all the features!
 - More features = more weights = model complexity increases = more ability to fit to training data
- Add some noise to the data.
- Specific to this algorithm, we can:
 - Implement **stochastic mini-batching**
 - Use our validation set to help decide when to stop training
 - Could keep an eye on generalization during training process
 - If we see that we're losing generalization, then we can say to *stop* the training process.
 - We could add a regularization term to our objective function to penalize complexity.


Note on Gradient Based Learning

Gradient learning is the basis of deep learning!

Resources

why minimize loss function instead of maximizing reward function?


Why is the "de-facto" in statistics to minimize the sum of squared errors cost function instead of maximizing some reward function like the likelihood function?

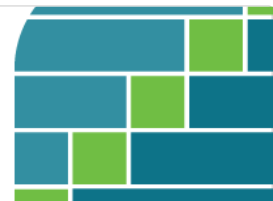
 <https://stats.stackexchange.com/questions/94783/why-minimize-loss-function-instead-of-maximizing-reward-function>



Why do we minimize the negative likelihood if it is equivalent to maximization of the likelihood?

This question has puzzled me for a long time. I understand the use of 'log' in maximizing the likelihood so I am not asking about 'log'.

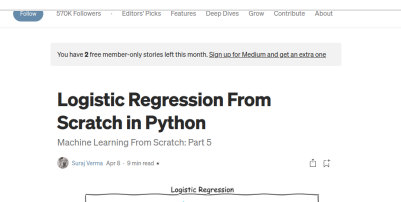
 <https://stats.stackexchange.com/questions/141087/why-do-we-minimize-the-negative-likelihood-if-it-is-equivalent-to-maximization-o>



Logistic Regression From Scratch in Python | by Suraj Verma | Towards...

archived 19 Aug 2021 15:27:28 UTC

 <https://archive.ph/hHWzO>



IBM Developer

 <https://developer.ibm.com/articles/implementing-logistic-regression-from-scratch-in-python/>