Logistic regression

April 25, 2023

1 Logistic regression

Logistic regression can serve as a stepping stone towards neural network algorithms and supervised deep learning. For logistic learning, the minimization of the cost function leads to a non-linear equation in the parameters β. The optimization of the problem calls therefore for minimization algorithms. This forms the bottleneck of all machine learning algorithms, namely how to find reliable minima of a multi-variable function. This leads us to the family of gradient descent methods. The latter are the workhorses of all modern machine learning algorithms.

1.0.1 Basics

We consider the case where the outcome y_i are discrete and only take values from $k = 0, \dots, K-1$ (i.e. K classes).

The goal is to predict the output classes from the design matrix $X \in \mathbb{R}^{n\times p}$ made of n samples, each of which carries p features or predictors. The primary goal is to identify the classes to which new unseen samples belong.

Let us specialize to the case of two classes only, with outputs $y_i = 0$ and $y_i = 1$. Our outcomes could represent the status of a credit card user that could default or not on her/his credit card debt.

One simple way to get a discrete output is to have sign functions that map the output of a linear regressor to values $\{0, 1\}$, $f(s_i) = sign(s_i) = 1$ if $s_i \geq 0$ and 0 if otherwise. We will encounter this model in our first demonstration of neural networks. Historically it is called the "perceptron" model in the machine learning literature. This model is extremely simple. However, in many cases it is more favorable to use a "soft" classifier that outputs the probability of a given category. This is achieved by the logistic function.

1.0.2 The logistic (or the fermi) function

In most classification models it is favorable to have a "soft" classifier that outputs the probability of a given category rather than a single value. For example, given x_i, the classifier outputs the probability of being in a category k. Logistic regression is the most common example of a so-called soft classifier. In logistic regression, the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the so-called logit function (or the fermi function) which is meant to represent the likelihood for a given event,

$$p(t) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t} = f(-t).$$

Many times we also use $\tanh(x)$ function, which is related to the fermi function by $\tanh(x) = f(-x) - f(x)$

We assume now that we have two outputs with y_i either 0 or 1. Furthermore we assume also that we have only two parameters β to fit. We define probabilities for different outcomes as follows

$$p(y_i = 1|x_i, \beta) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)}, p(y_i = 0|x_i, \beta) = 1 - p(y_i = 1|x_i, \beta),$$

where β are the weights we wish to extract from data, in our case β_0 and β_1.

Here we read $p(y_i = 1|x_i, \beta)$ as probability for the outcome $y_i = 1$ given input variables x_i and given fixed parameters β.

Note that we used

$$p(y_i = 0|x_i, \beta) + p(y_i = 1|x_i, \beta) = 1.$$

At the output we will define that $\tilde{y}_i = 1$ if $p(y_i = 1|x_i, \beta) > 1/2$ and is 0 otherwise.

1.1 Maximum likelihood

In order to define the total likelihood for all possible outcomes from a
dataset $\mathcal{D} = \{(y_i, x_i)\}$, with the binary labels $y_i \in \{0, 1\}$ and where the data points are drawn independently, we use the so-called Maximum Likelihood Estimation (MLE) principle.

We aim thus at maximizing the probability of seeing the observed data. We can then approximate the likelihood in terms of the product of the individual probabilities of a specific outcome y_i, that is

$$P(\mathcal{D}|\beta) = \prod_{i=1}^{n} \left[p(y_i = 1|x_i, \beta)\right]^{y_i} \left[1 - p(y_i = 1|x_i, \beta))\right]^{1-y_i}$$

If this probability is maximized, its log value is also maximized, and $-\log$ value is minimized. We can then define a corresponding **cost** function as

$$C(\beta) = -\log(P(\mathcal{D}|\beta))$$

$$\mathcal{C}(\beta) = \sum_{i=1}^{n} -y_i \log[p(y_i = 1|x_i, \beta)] - (1 - y_i) \log\left[1 - p(y_i = 1|x_i, \beta))\right].$$

This equation is known in statistics as the **cross entropy**.

Finally, we note that just as in linear regression, in practice we often supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization as we did for Ridge and Lasso regression.

1.2 Minimizing the cross entropy cost function

The derivative can be easily derived

$$\frac{\partial C(\beta)}{\partial \beta_j} = \sum_{i=1}^{n} \left(-\frac{y_i}{p_i} + \frac{1 - y_i}{1 - p_i}\right) \frac{\partial p_i}{\partial \beta_j} \tag{1}$$

where we denoted $p(y_i = 1|x_i, \beta) = p_i$ and

$$\frac{\partial p_i}{\partial \beta_j} = \frac{\partial}{\partial \beta_j}\frac{1}{1+e^{-(\beta_0+\beta_1 x_i)}} = \frac{1}{1+e^{-(\beta_0+\beta_1 x_i)}}\frac{1}{1+e^{(\beta_0+\beta_1 x_i)}}(\delta_{j=0}+x_i\delta_{j=1}) = p_i(1-p_i)(\delta_{j=0}+x_i\delta_{j=1}) \tag{2}$$

We therefore see that

$$\frac{\partial C(\beta)}{\partial \beta_j} = \sum_{i=1}^{n}\left(-\frac{y_i}{p_i}+\frac{1-y_i}{1-p_i}\right)p_i(1-p_i)(\delta_{j=0}+x_i\delta_{j=1}) = \sum_{i=1}^{n}(p_i-y_i)(\delta_{j=0}+x_i\delta_{j=1}) \tag{3}$$

If we defined a design matrix with two columns $X^T = [1, x_i]$, i.e., the first column is 1 and the second is the input observable x_i, than we can write the derivative of the cost function as

$$\frac{\partial C(\beta)}{\partial \beta_j} = \mathbf{X}^T(\mathbf{p}-\mathbf{y})$$

Here $\mathbf{p}, \mathbf{y} \in \mathbb{R}^n$ and $\mathbf{X} \in \mathbb{R}^{n\times 2}$

The second derivative of the cost function is

$$\frac{\partial^2 C(\beta)}{\partial \beta_l \partial \beta_j} = \frac{\partial}{\partial \beta_l}\sum_{i=1}^{n}(p_i-y_i)(\delta_{j=0}+x_i\delta_{j=1}) = \sum_{i=1}^{n}(\delta_{j=0}+x_i\delta_{j=1})\frac{\partial p_i}{\partial \beta_l} = \sum_{i=1}^{n}(\delta_{j=0}+x_i\delta_{j=1})p_i(1-p_i)(\delta_{l=0}+x_i\delta_{l=1}) \tag{4}$$

This can be compactly written as

$$\frac{\partial^2 C(\beta)}{\partial \beta^T \partial \beta} = \mathbf{X}^T\mathbf{W}\mathbf{X} \tag{5}$$

where W is a diagonal matrix with entries $W_{ii} = p_i(1-p_i)$.

This matrix is positive definite, which can be proven by noting that we can redefine $\widetilde{X}_{ij} = \sqrt{p_i(1-p_i)}X_{ij}$, in terms of which the second derivative is just $\widetilde{X}^T\widetilde{X}$. Than the proof is identical to the one in linear regression. Namely, the singular values of $\widetilde{X}$ exist and are σ, therefore the eigenvalues of $\widetilde{X}^T\widetilde{X}$ are equal to σ^2, and therefore the matrix is positive definite.

The consequence is that the global minimum exists and the cost function is minimum in the solution

$$\frac{\partial C(\beta)}{\partial \beta} = 0$$

.

Finally, to find the minimum with respect to the parameters β_0, β_1, we need to solve nonlinear set of equations, which can only be done numerically.

The method of choice is the **Newton's method**, or, in multidimensional case we also call it the **gradien descent** method.

1.2.1 Gradient descent method

If we want to find zero of a gradient

$$0 = \frac{\partial C}{\partial \beta_j} = \sum_i X_{ji}(p_i(\beta)-y_i) \equiv g_j(\beta)$$

we can Taylor expand around $g_j(\beta) = 0$, which gives us the famous Neton's-Raphson method

$$\beta_{k+1} = \beta_k - (\nabla_\beta g(\beta_k))^{-1} g(\beta_k)$$

which is a generalization of the 1D case $x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)}$. Note that $g(\beta_k)$ is a vector, and its derivative is a matrix.

We note that

$$\nabla_\beta g(\beta_k) \equiv \frac{\partial^2 C(\beta)}{\partial \beta_l \partial \beta_j}$$

is the second derivative and is the so-called Hessian matrix. We will denote it by

$$H_{lj}(\beta) \equiv \frac{\partial^2 C(\beta)}{\partial \beta_l \partial \beta_j}$$

In the Newton's method we thus need to iterate the following equation

$$\beta_{k+1} = \beta_k - H^{-1}(\beta_k) g(\beta_k)$$

In most practical applications, the Hessian matrix is very expensive to calculate and to invert. Instead, in most gradient descent methods one approximates H by a number, which is called **the learning rate**.

The later is usually a parameter that is being modified or adjusted with iterations, so that minimum of C is found. Note that the equation

$$\beta_{k+1} = \beta_k - \gamma g(\beta_k)$$

for appropriate number $\gamma > 0$ is also approaching the minimum, as $g(\beta)$ is gradient in the direction of descent of function $C(\beta)$. The problem is that we don't know apriory how large should γ be.

These equations are usually termed **gradient descent (GD)** method optimizing function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$.

It can be shown that if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k),$$

with $\gamma_k > 0$. For γ_k small enough, then $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum.

In machine learing we are often faced with non-convex high dimensional cost functions with many local minima. Since **GD** is deterministic we will get stuck in a local minimum, if the method converges, unless we have a very good intial guess. This also implies that the scheme is sensitive to the chosen initial condition.

Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD).

1.3 Stochastic Gradient Descent

The underlying idea of SGD comes from the observation that the cost function, which we want to minimize, can almost always be written as a sum over n data points $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\ ) = \sum_{i=1}^{n} c_i(\mathbf{x}_i,\ ).$$

This in turn means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\ ) = \sum_{i=1}^{n} \nabla_\beta c_i(\mathbf{x}_i,\ ).$$

Stochasticity/randomness is introduced by only taking the gradient on a subset of the data called minibatches. If there are n data points and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$. We have

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i,\ )$$

where k is picked at random with equal probability from $[1, n/M]$. An iteration over the number of minibathces (n/M) is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches.

More complex optimizers are used in ML, in particular ADAM, which are much more technical and beyond the scope of these lectures.

1.4 Extending to more classes (beyond two outputs)

If the otput requires more than two outpus, lets say K (exmple: digits between $0-9$), we can generalized the sigmoid function to so called softmax function, in which each ouput neuron has probability proportional to e^{z_j} with $j \in [0, .., K-1]$. The function for probability would have the following form:

$$f(z_i) = \frac{\exp(z_i)}{\sum_{m=1}^{K} \exp(z_m)}.$$

where $z_i = \sum_j \beta_{ij} x_j + b_i$.

This function is called **Softmax function**, which is actually Boltzman weighted sum.

Note that the derivatives are now a bit more involved, i.e.,

$$\frac{\partial f(z_i)}{\partial \beta_{jk}} = \frac{\exp(z_i)}{\sum_{m=1}^{K} \exp(z_m)} \frac{\partial z_i}{\partial \beta_{jk}} - \sum_p \frac{\exp(z_i)}{(\sum_{m=1}^{K} \exp(z_m))^2} \exp(z_p) \frac{\partial z_p}{\partial \beta_{jk}}.$$

Since

$$\frac{\partial z_p}{\partial \beta_{jk}} = \delta_{p,j} x_k$$

we have

$$\frac{\partial f(z_i)}{\partial \beta_{jk}} = f(z_i)\left(\delta_{ij} - f(z_j)\right) x_k \frac{\partial f(z_i)}{\partial b_j} = f(z_i)\left(\delta_{ij} - f(z_j)\right),$$

which in case of the simply binary model reduces to having $i = j$.

[ ]: