Multiclass Logistic Regression, Multiplayer Perceptron

Milan Straka

November 11, 2019
Logistic Regression

An extension of perceptron, which models the conditional probabilities of $p(C_0|x)$ and of $p(C_1|x)$. Logistic regression can in fact handle also more than two classes, which we will see shortly.

Logistic regression employs the following parametrization of the conditional class probabilities:

$$P(C_1|x) = \sigma(x^t w + b)$$
$$P(C_0|x) = 1 - P(C_1|x),$$

where $\sigma$ is a *sigmoid function*

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$ 

Can be trained using an SGD algorithm.
The sigmoid function has values in range \((0, 1)\), is monotonically increasing and it has a derivative of \(\frac{1}{4}\) at \(x = 0\).

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]

\[
\sigma'(x) = \sigma(x)(1 - \sigma(x))
\]
Logistic Regression

To give some meaning to the sigmoid function, starting with

\[ P(C_1|\mathbf{x}) = \sigma(f(\mathbf{x}; \mathbf{w})) = \frac{1}{1 + e^{-f(\mathbf{x}; \mathbf{w})}} \]

we can arrive at

\[ f(\mathbf{x}; \mathbf{w}) = \log \left( \frac{P(C_1|\mathbf{x})}{P(C_0|\mathbf{x})} \right), \]

where the prediction of the model \( f(\mathbf{x}; \mathbf{w}) \) is called a logit and it is a logarithm of odds of the two classes probabilities.
Logistic Regression

To train the logistic regression $y(x; w) = x^T w$, we use MLE (the maximum likelihood estimation). Note that $P(C_1|x; w) = \sigma(y(x; w))$.

Therefore, the loss for a batch $X = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$ is

$$L(X) = \frac{1}{N} \sum_i - \log(P(C_{t_i}|x; w)).$$

**Input:** Input dataset $(X \in \mathbb{R}^{N \times D}, t \in \{0, +1\})$, learning rate $\alpha \in \mathbb{R}^+$.  

- $w \leftarrow 0$
- until convergence (or until patience is over), process batch of $N$ examples:
  - $g \leftarrow -\frac{1}{N} \sum_i \nabla_w \log(P(C_{t_i}|x; w))$
  - $w \leftarrow w - \alpha g$
Linearity in Logistic Regression

Figure 4.12 of Pattern Recognition and Machine Learning.
Multiclass Logistic Regression

To extend the binary logistic regression to a multiclass case with $K$ classes, we:

- Generate multiple outputs, notably $K$ outputs, each with its own set of weights, so that
  \[ y(x; W)_i = W_i x. \]

- Generalize the sigmoid function to a softmax function, such that
  \[ \text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}. \]

Note that the original sigmoid function can be written as

\[ \sigma(x) = \text{softmax} \left( \begin{bmatrix} x & 0 \end{bmatrix} \right)_0 = \frac{e^x}{e^x + e^0} = \frac{1}{1 + e^{-x}}. \]

The resulting classifier is also known as \textit{multinomial logistic regression}, \textit{maximum entropy classifier} or \textit{softmax regression}. 

Multiclass Logistic Regression

Note that as defined, the multiclass logistic regression is overparametrized. It is possible to generate only $K - 1$ outputs and define $z_K = 0$, analogously to binary logistic regression.

In this settings, analogously to binary logistic regression, we can recover the interpretation of softmax inputs as logits:

$$\text{softmax}(z)_i = \log \left( \frac{P(C_i | x; w)}{P(C_K | x; w)} \right).$$

However, in all our implementations, we will use weights for all $K$ outputs.
Multiclass Logistic Regression

Using the softmax function, we naturally define that

\[ P(C_i|x; W) = \text{softmax}(W_i x)_i = \frac{e^{W_i x}}{\sum_j e^{W_j x}}. \]

We can then use MLE and train the model using stochastic gradient descent.

**Input:** Input dataset \((X \in \mathbb{R}^{N \times D}, t \in \{0, 1, \ldots, K - 1\})\), learning rate \(\alpha \in \mathbb{R}^+\).

1. \(w \leftarrow 0\)
2. until convergence (or until patience is over), process batch of \(N\) examples:
   1. \(g \leftarrow -\frac{1}{N} \sum_i \nabla_w \log(P(C_{ti}|x; w))\)
   2. \(w \leftarrow w - \alpha g\)
Multiclass Logistic Regression

Note that the decision regions of the binary/multiclass logistic regression are singly connected and convex.

To see this, consider $\mathbf{x}_A$ and $\mathbf{x}_B$ in the same decision region $R_k$.

Any point $\mathbf{x}$ lying on the line connecting them is their linear combination, $\mathbf{x} = \lambda \mathbf{x}_A + (1 - \lambda) \mathbf{x}_B$, and from the linearity of $y(\mathbf{x}) = \mathbf{W} \mathbf{x}$ it follows that

$$y(\mathbf{x}) = \lambda y(\mathbf{x}_A) + (1 - \lambda) y(\mathbf{x}_B).$$

Given that $y_k(\mathbf{x}_A)$ was the largest among $y(\mathbf{x}_A)$ and also given that $y_k(\mathbf{x}_B)$ was the largest among $y(\mathbf{x}_B)$, it must be the case that $y_k(\mathbf{x})$ is the largest among all $y(\mathbf{x})$. 

Figure 4.3 of Pattern Recognition and Machine Learning.
Mean Square Error as MLE

During regression, we predict a number, not a real probability distribution. In order to generate a distribution, we might consider a distribution with the mean of the predicted value and a fixed variance $\sigma^2$ – the most general such a distribution is the normal distribution.
Mean Square Error as MLE

Therefore, assume our model generates a distribution

\[ P(y|x; w) = \mathcal{N}(y; f(x; w), \sigma^2). \]

Now we can apply MLE and get

\[
\arg\max_{w} P(X; w) = \arg\min_{w} \sum_{i=1}^{m} - \log P(y_i|x_i; w)
\]

\[
= - \arg\min_{w} \sum_{i=1}^{m} \log \sqrt{\frac{1}{2\pi\sigma^2}} e^{-\frac{(y_i-f(x_i;w))^2}{2\sigma^2}}
\]

\[
= - \arg\min_{w} m \log(2\pi\sigma^2)^{-1/2} + \sum_{i=1}^{m} -\frac{(y_i-f(x_i;w))^2}{2\sigma^2}
\]

\[
= \arg\min_{w} \sum_{i=1}^{m} \frac{(y_i - f(x_i; w))^2}{2\sigma^2} = \arg\min_{w} \sum_{i=1}^{m}(y_i - f(x_i; w))^2.
\]
Multilayer Perceptron

Input layer

Hidden layer

Output layer

\[ x_1 \rightarrow h_1 \]

\[ x_2 \rightarrow h_2 \]

\[ x_3 \rightarrow h_3 \]

\[ x_4 \rightarrow h_4 \]

\[ h_1 \rightarrow o_1 \]

\[ h_2 \rightarrow o_1 \]

\[ h_3 \rightarrow o_2 \]

\[ h_4 \rightarrow o_2 \]
Multilayer Perceptron

There is a weight on each edge, and an activation function $f$ is performed on the hidden layers, and optionally also on the output layer.

\[
h_i = f \left( \sum_j w_{i,j} x_j + b_i \right)
\]

If the network is composed of layers, we can use matrix notation and write:

\[
h = f \left( W x + b \right)
\]
Multilayer Perceptron and Biases

Figure 5.1 of Pattern Recognition and Machine Learning.
**Output Layers**

- none (linear regression if there are no hidden layers)

\[ \sigma(x) \overset{\text{def}}{=} \frac{1}{1 + e^{-x}} \]

- softmax (maximum entropy model if there are no hidden layers)

\[ \text{softmax}(x) \propto e^{x} \]

\[ \text{softmax}(x)_i \overset{\text{def}}{=} \frac{e^{x_i}}{\sum_j e^{x_j}} \]
Neural Network Activation Functions

Hidden Layers

- none (does not help, composition of linear mapping is a linear mapping)
- $\sigma$ (but works badly – nonsymmetrical, $\frac{d\sigma}{dx}(0) = 1/4$)
- tanh
  - result of making $\sigma$ symmetrical and making derivation in zero 1
  - $\tanh(x) = 2\sigma(2x) - 1$
- ReLU
  - $\max(0, x)$
The multilayer perceptron can be trained using an SGD algorithm:

**Input:** Input dataset \((\mathbf{X} \in \mathbb{R}^{N \times D}, \mathbf{t} \in \{0, +1\})\), learning rate \(\alpha \in \mathbb{R}^+\).

- \(w \leftarrow 0\)
- until convergence (or until patience is over), process batch of \(N\) examples:
  - \(g \leftarrow \nabla w \frac{1}{N} \sum_j -\log p(y_j | \mathbf{x}_j ; \mathbf{w})\)
  - \(w \leftarrow w - \alpha g\)
Training MLP – Computing the Derivatives

Assume a network with an input of size $N_1$, then weights $U \in \mathbb{R}^{N_1 \times N_2}$, hidden layer with size $N_2$ and activation $h$, weights $V \in \mathbb{R}^{N_2 \times N_3}$, and finally an output layer of size $N_3$ with activation $o$. 
(to be finished later)
Universal Approximation Theorem '89

Let \( \varphi(x) \) be a nonconstant, bounded and monotonically-increasing continuous function. (Later a proof was given also for \( \varphi = \text{ReLU} \).)

Then for any \( \varepsilon > 0 \) and any continuous function \( f \) on \([0, 1]^m\) there exists an \( N \in \mathbb{N}, v_i \in \mathbb{R}, b_i \in \mathbb{R} \) and \( \omega_i \in \mathbb{R}^m \), such that if we denote

\[
F(x) = \sum_{i=1}^{N} v_i \varphi(\omega_i \cdot x + b_i)
\]

then for all \( x \in [0, 1]^m \)

\[
|F(x) - f(x)| < \varepsilon.
\]
Universal Approximation Theorem for ReLUs

Sketch of the proof:

- If a function is continuous on a closed interval, it can be approximated by a sequence of lines to arbitrary precision.

\[ n_1(x) = \text{ReLU}(-5x - 7.7) \]
\[ n_2(x) = \text{ReLU}(-1.2x - 1.3) \]
\[ n_3(x) = \text{ReLU}(1.2x + 1) \]
\[ n_4(x) = \text{ReLU}(1.2x - 0.2) \]
\[ n_5(x) = \text{ReLU}(2x - 1.1) \]
\[ n_6(x) = \text{ReLU}(5x - 5) \]

\[ Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x) \]

However, we can create a sequence of \( k \) linear segments as a sum of \( k \) ReLU units – on every endpoint a new ReLU starts (i.e., the input ReLU value is zero at the endpoint), with a tangent which is the difference between the target tangent and the tangent of the approximation until this point.
Universal Approximation Theorem for Squashes

Sketch of the proof for a squashing function $\varphi(x)$ (i.e., nonconstant, bounded and monotonically-increasing continuous function like sigmoid):

- We can prove $\varphi$ can be arbitrarily close to a hard threshold by compressing it horizontally.

\[
y = \frac{1}{1 + e^{-(w^T x + b)}}
\]

- Then we approximate the original function using a series of straight line segments.

https://hackernoon.com/hn-images/1*N7dfPwbiXC-Kk4TCbfRerA.png

https://hackernoon.com/hn-images/1*hVuJgUTLUFWTMmJhl_fomg.png

https://hackernoon.com/hn-images/1*hVUJgUTLUFWTMmJhL_fomg.png
Given a function $J(x)$, we can find a maximum with respect to a vector $x \in \mathbb{R}^d$, by investigating the critical points $\nabla_x J(x) = 0$.

Consider now finding maximum subject to a constraint $g(x) = 0$.

- Note that $\nabla_x g(x)$ is orthogonal to the surface of the constraining, because if $x$ and a nearby point $x + \varepsilon$ lie on the surface, from the Taylor expansion $g(x + \varepsilon) \approx g(x) + \varepsilon^T \nabla_x g(x)$ we get $\varepsilon^T \nabla_x g(x) \approx 0$.

- In the sought maximum, $\nabla_x f(x)$ must also be orthogonal to the constraining surface (or else moving in the direction of the derivative would increase the value).

- Therefore, there must exist $\lambda$ such that $\nabla_x f + \lambda \nabla_x g = 0$. 

Figure E.1 of Pattern Recognition and Machine Learning.
We therefore introduce the *Lagrangian function*

\[ L(x, \lambda) \overset{\text{def}}{=} f(x) + \lambda g(x). \]

We can then find the maximum under the constraint by inspecting critical points of \( L(x, \lambda) \) with respect to both \( x \) and \( \lambda \):

- \( \frac{\partial L}{\partial \lambda} = 0 \) leads to \( g(x) = 0 \);
- \( \frac{\partial L}{\partial x} = 0 \) is the previously derived \( \nabla_x f + \lambda \nabla_x g = 0 \).
Calculus of Variations

Many optimization techniques depend on minimizing a function $J(w)$ with respect to a vector $w \in \mathbb{R}^d$, by investigating the critical points $\nabla_w J(w) = 0$.

A function of a function, $J[f]$, is known as a **functional**, for example entropy $H[\cdot]$.

Similarly to partial derivatives, we can take **functional derivatives** of a functional $J[f]$ with respect to individual values $f(x)$ for all points $x$. The functional derivative of $J$ with respect to a function $f$ in a point $x$ is denoted as

$$\frac{\partial}{\partial f(x)} J.$$

For this class, we will use only the following theorem, which states that for all differentiable functions $f$ and differentiable functions $g(y, x)$ with continuous derivatives, it holds that

$$\frac{\partial}{\partial f(x)} \int g(f(x), x) \, dx = \frac{\partial}{\partial y} g(f(x), x).$$
Calculus of Variations

An intuitive view is to think about $f(x)$ as a vector of uncountably many elements (for every value $x$). In this interpretation the result is analogous to computing partial derivatives of a vector $w \in \mathbb{R}^d$:

$$\frac{\partial}{\partial w_i} \sum_j g(w_j, x) = \frac{\partial}{\partial w_i} g(w_i, x).$$

$$\frac{\partial}{\partial f(x)} \int g(f(x), x) \, dx = \frac{\partial}{\partial y} g(f(x), x).$$
Function with Maximum Entropy

What distribution over \( \mathbb{R} \) maximizes entropy \( H[p] = -\mathbb{E}_x \log p(x) \)?

For continuous values, the entropy is an integral \( H[p] = -\int p(x) \log p(x) \, dx \).

We cannot just maximize \( H \) with respect to a function \( p \), because:

- the result might not be a probability distribution – we need to add a constraint that \( \int p(x) \, dx = 1 \);
- the problem is unspecified because a distribution can be shifted without changing entropy – we add a constraint \( \mathbb{E}[x] = \mu \);
- because entropy increases as variance increases, we ask which distribution with a \textit{fixed} variance \( \sigma^2 \) has maximum entropy – adding a constraint \( \text{Var}(x) = \sigma^2 \).
Lagrangian of all the constraints and the entropy function is
\[
L(p; \mu, \sigma^2) = \lambda_1 \left( \int p(x) \, dx - 1 \right) + \lambda_2 (\mathbb{E}[x] - \mu) + \lambda_3 \left( \text{Var}(x) - \sigma^2 \right) + H[p].
\]

By expanding all definitions to integrals, we get
\[
L(p; \mu, \sigma^2) = \int \left( \lambda_1 p(x) + \lambda_2 p(x)x \lambda_3 p(x)(x - \mu)^2 - p(x) \log p(x) \right) \, dx - \lambda_1 - \mu \lambda_2 - \sigma^2 \lambda_3.
\]

The functional derivative of \( L \) is:
\[
\frac{\partial}{\partial p(x)} L(p; \mu, \sigma^2) = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0.
\]
Function with Maximum Entropy

Rearranging the functional derivative of $L$:

\[
\frac{\partial}{\partial p(x)} L(p; \mu, \sigma^2) = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0.
\]

we obtain

\[
p(x) = \exp \left( \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 \right).
\]

We can verify that setting $\lambda_1 = 1 - \log \sigma \sqrt{2\pi}$, $\lambda_2 = 0$ and $\lambda_3 = -1/(2\sigma^2)$ fulfils all the constraints, arriving at

\[
p(x) = \mathcal{N}(x; \mu, \sigma^2).
\]