Beta Regression vs. Logistic Regression

References:

1. Peter Sadowski and Pierre Baldi, Neural Network Regression with Beta, Dirichlet, and Dirichlet-Multinomial Outputs, *ICLR*, 2019
2. Silvia Ferrari and Francisco Cribari-NetoBeta Regression for Modelling Rates and Proportions, *Journal of Applied Statistics*, 31:7, 799-815, DOI: 10.1080/0266476042000214501, 2004

Pre-requisites:

1. Knowing the Maximum-likelihood criterion;
2. It is recommended that readers have known the concepts and basic applications of Beta distribution.
3. There will be a recap on how Maximum-likelihood criterion is applied to do regression (example from image classifier). But readers are recommended to have known such a process. (Anyway, I am going to try my best to explain that.)

Written by Tongyu Lu, March 15, 2021

Contents:

Logistic Regression - Recap

Assume that we want to do image classification between cats and dogs.
Each input tensor (image) is denoted as $x \in [0, 256]^{B\times W\times C}$, and each output is denoted as $y \in \{0, 1\}$.
There corresponding random variables (RVs) are denoted by X and Y.
Our classifier is denoted as a function $g(\cdot|\theta) : [0, 256]^{B\times W\times C} \to [0, 1]$, which predicts the probability of the image being a dog. Maybe this classifier is a CNN, but we do not really care about that right now; all we need to know is that the classifier is controlled by parameter θ, which is to be optimized. Normally, we denote $g(x|\theta)$ as $\hat{y}$.

What is the optimal parameter θ? Normally, we use the maximum-likelihood criterion to find that. How comes?

We start from the distribution of Y: it is Bernoulli because it is a either-or choice. Assume that the probability of "image x represents a dog" or "$Y = 1$ given x" is $\beta(x)$. Then, we say that
$P(Y = y|x) = \beta(x)^y(1 - \beta(x))^{1-y}, y = 0, 1$.

Now, we want to see if $\beta(x)$ gives the right answer. Actually, $\hat{y} = \beta(x) = g(x|\theta)$. To deal with this, we calculate the log-likelihood term

$$l(\theta) = \log(P(Y = y|x)) = y\log\beta(x) + (1 - y)\log(1 - \beta(x)) = y\log g(x|\theta) + (1 - y)\log(1 - g(x|\theta))$$.

When we observe a bunch of data $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, we define the log-likelihood term as $L(\theta) = \sum_{k=1}^{N} y_k \log g(x_k|\theta) + (1 - y_k) \log(1 - g(x_k|\theta))$. All we want is to maximize the likelihood by adjusting parameter θ.

The causal line for our model is that: "θ controls β when given x, β controls Y which gives the final result".

Therefore, in the language of machine learning, we define the loss function as $\text{loss}(\theta) = -L(\theta) = -\sum_{k=1}^{N} y_k \log g(x_k|\theta) + (1 - y_k) \log(1 - g(x_k|\theta))$. And we solve the unconstrained argmin problem $\theta^* = \arg\min_\theta \text{loss}(\theta)$ by back-propagation.

And we say that this is a logistic regression, when $\hat{y} = g(x|\theta) = \phi(f(x|\theta))$, where $\phi(z)$ is sigmoid function defined as $\phi(z) = \frac{1}{1+e^{-z}}$. In this case, we could find that (after calculation) $l(\theta) = (1 - y)f(x|\theta) - \hat{y}$, which is a pretty simple form compared to the "log" form.

Beta Regression - A Similar but Different Case

In the classification example, we assumed that the distribution of classifier output is Bernoulli. But actually, our output situates between 0 and 1.

Can this model be used for regressing variables which situate between 0 and 1, but do not obey Bernoulli distribution?

Of course we can, because the domain of our model output (between 0 and 1) is the same as the desired one. However, it is obvious that Bernoulli distribution is not tailored for a continuous random variable. In other words, there is something wrong with our loss function.

Recall that we define the loss function according maximum-likelihood criterion, which looks like: $\text{loss}(\theta) = -L(\theta) = -\sum_{k=1}^{N} \log p_Y(y_k|\beta(x_k))$, where β serves as the parameter of the distribution of Y and is determined by the input x.

The key is on the function p_Y: in the classification case, p_Y is binomial (Bernoulli) and discrete, but it could be other distributions.
Recall the Beta distribution: it is also suitable to model RVs which situate between 0 and 1! Here is a link to my introduction to Beta distribution and its applications on modeling percentages and model parameters: From_Beta_Distribution_to_Conjugate_Distributions.

Let us have a brief comparison between Bernoulli discrete RV and Beta continuous RV:

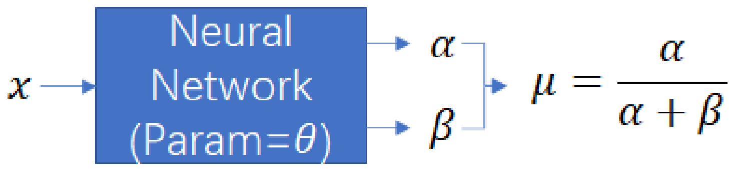
- (Bernoulli) $Y \sim B(\beta) \Leftrightarrow P(Y = y) = \beta^y (1 - \beta)^{1-y}, y = 0, 1, \beta \in [0, 1]$
- (Beta) $Y \sim Beta(\alpha, \beta) \Leftrightarrow p_Y(y|\alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)+\Gamma(\beta)} y^{\alpha-1}(1 - y)^{\beta-1}, y \in [0, 1], \alpha, \beta > 0$

Did you get it? Alternatively, we could model p_Y as Beta distribution. And now our likelihood function becomes

$$l(\alpha, \beta) = \log p_Y(y|\alpha, \beta) = \log[\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)+\Gamma(\beta)} y^{\alpha-1}(1 - y)^{\beta-1}] = (\alpha - 1) \log y + (\beta - 1) \log(1 - y) + \log \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)+\Gamma(\beta)}$$

.
Alternatively, we could replace α, β with parameter μ, γ, where $\alpha = \mu\gamma$ and $\beta = (1 - \mu)\gamma$. Parameter $\mu \in (0, 1)$ is the mean, which is suitable for model output.

The role of our Beta-neural-network might be estimating α, β, and calculate the final output as $\mu = \alpha/(\alpha+\beta)$. The loss function could be treated as $\text{loss}(\theta) = -L(\theta) = -\sum_{k=1}^{N} \log p_Y(y_k|\alpha(x_k), \beta(x_k))$.
The feed-forward process may look like this:



x
Neural Network (Param=θ)
α
β
$\mu = \frac{\alpha}{\alpha+\beta}$

Notice that $\alpha, \beta > 0$. Therefore, we could output them with a simple non-negative activate function.

An Experiment for Beta Regression

I want to approximate $f : \mathbf{R}^6 \to [0, 1]$ which is defined as $f(x) = 0.5\cos(|\bar{x}|^{1.5} + 0.25\bar{x}^2) + 0.5$, where $\bar{x} = \frac{1}{6}\sum_{k=1}^{6} x_k$.

And I defined three NNs to approximate it:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data.dataset import Dataset
from torch.utils.data import DataLoader

import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline

class BetaNN(nn.Module):
    def __init__(self, mid=10):
        super(BetaNN, self).__init__()
        self.fc1 = nn.Linear(6, mid)
        self.fc2 = nn.Linear(mid, mid)
        self.fc3 = nn.Linear(mid, mid)
        self.fc_a = nn.Linear(mid, 1)
        self.fc_b = nn.Linear(mid, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = torch.cos(self.fc3(x))
        a = F.softplus(self.fc_a(x)).squeeze(1)
        b = F.softplus(self.fc_b(x)).squeeze(1)
        y = (a/(a+b))
        gamma = a+b
        return y,gamma

class LinearNN(nn.Module):
    def __init__(self, mid=10):
        super(LinearNN, self).__init__()
        self.fc1 = nn.Linear(6, mid)
        self.fc2 = nn.Linear(mid, mid)
        self.fc3 = nn.Linear(mid, mid)
        self.fcy = nn.Linear(mid, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = torch.cos(self.fc3(x))
        y = F.relu(self.fcy(x)).squeeze(1)
        return y

class SigmoidNN(nn.Module):
    def __init__(self, mid=10):
        super(SigmoidNN, self).__init__()
        self.fc1 = nn.Linear(6, mid)
        self.fc2 = nn.Linear(mid, mid)
        self.fc3 = nn.Linear(mid, mid)
        self.fcy = nn.Linear(mid, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = torch.cos(self.fc3(x))
        y = F.sigmoid(self.fcy(x)).squeeze(1)
        return y
```

The dataset is defined as:

```
class VectorGenerator:
    def __init__(self, max_val = 10):
        self.max_val = np.abs(max_val)
    def gen_data(self):
        x1 = np.random.rand(2)*self.max_val*2-self.max_val
        x2 = np.random.rand(2)*self.max_val*2-self.max_val
        x3 = np.random.rand(2)*self.max_val*2-self.max_val
        x = np.concatenate((x1,x2,x3), axis=0).astype(np.float32)
        return x

class NonLinearFuncDataset(Dataset):
    def __init__(self, max_val = 10):
        self.generator = VectorGenerator(max_val = 10)

    def __getitem__(self, index):
        x = self.generator.gen_data()
        y = (np.cos(np.abs(np.mean(x))**1.5 + 0.25*np.mean(x)**2)+1)/2
        return x, y

    def __len__(self):
        return 1000000
```

Then, we design 4 experiments:

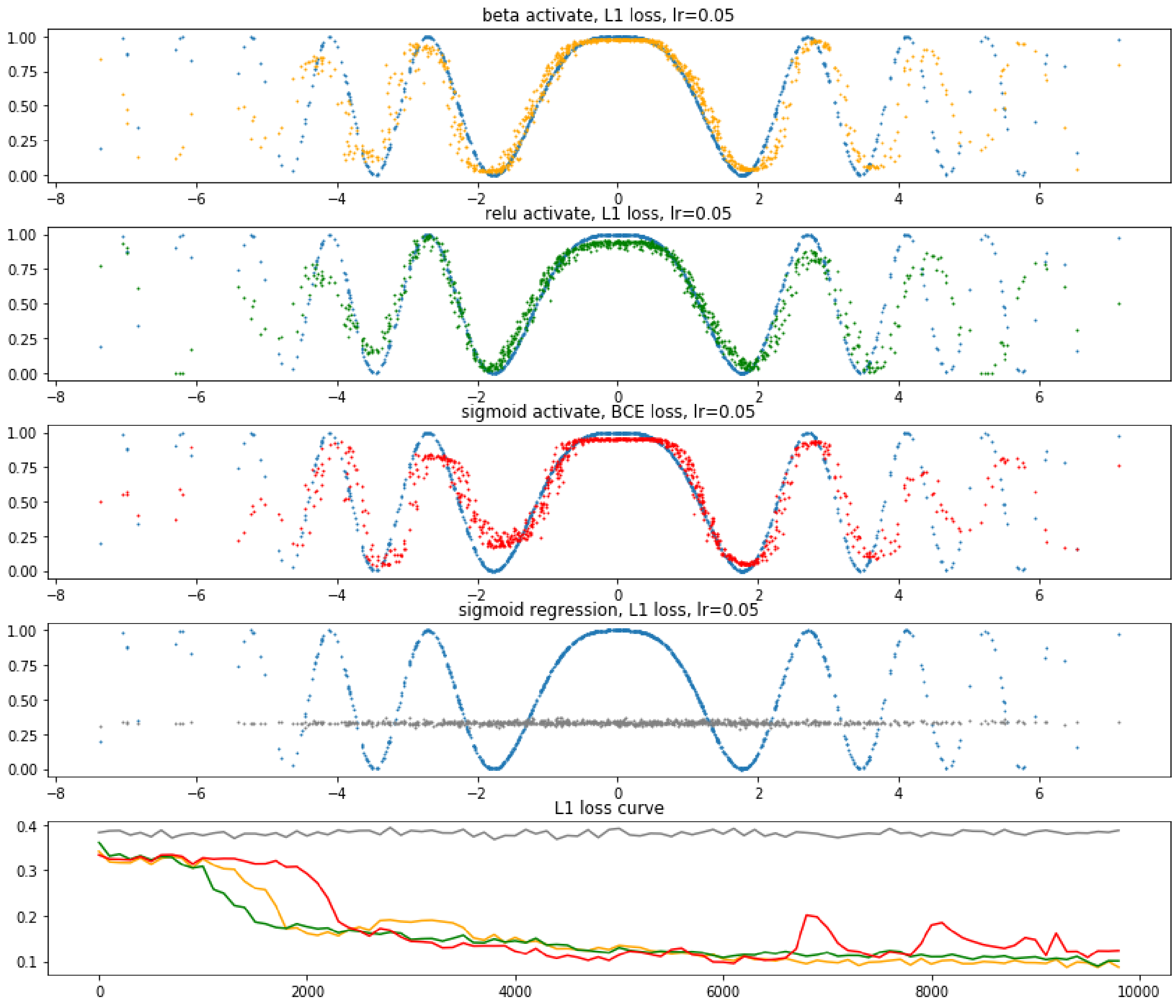
1. use L1 loss to train beta-activated nn
2. use L1 loss to train relu-activated nn

3. use BCE loss to train sigmoid-activated nn
4. use L1 loss to train sigmoid-activated nn

```
beta_nn = BetaNN(mid=10)
optimizer = optim.SGD(beta_nn.parameters(), lr=0.05)

linear_nn = LinearNN(mid=10)
optimizer2 = optim.SGD(linear_nn.parameters(), lr=0.05)

sigmoid_nn = SigmoidNN(mid=10)
optimizer3 = optim.SGD(sigmoid_nn.parameters(), lr=0.05)

sigmoid_nn_l1 = SigmoidNN(mid=10)
optimizer4 = optim.SGD(sigmoid_nn.parameters(), lr=0.05)

train_dataset = NonLinearFuncDataset(max_val = 3)
train_loader = DataLoader(dataset = train_dataset, batch_size = 12)
train_iter = iter(train_loader)

while 1:
    x, y_true = next(train_iter)

    beta_nn.train()
    y_hat, gamma = beta_nn(x)
    loss = F.l1_loss(y_hat, y_true)
    loss.backward()
    loss_buf += loss.detach()
    optimizer.step()
    optimizer.zero_grad()

    linear_nn.train()
    y_hat2 = linear_nn(x)
    loss2 = F.l1_loss(y_hat2, y_true)
    loss2.backward()
    loss_buf2 += loss2.detach()
    optimizer2.step()
    optimizer2.zero_grad()

    sigmoid_nn.train()
    y_hat3 = sigmoid_nn(x)
    loss3 = F.binary_cross_entropy(y_hat3, y_true)
    loss3.backward()
    loss_buf3 += F.l1_loss(y_hat3, y_true).detach()
    optimizer3.step()
    optimizer3.zero_grad()

    sigmoid_nn_l1.train()
    y_hat4 = sigmoid_nn_l1(x)
    loss4 = F.l1_loss(y_hat4, y_true)
    loss4.backward()
    loss_buf4 += loss4.detach()
    optimizer4.step()
    optimizer4.zero_grad()

    # hidden: print and update loss buffer
```

```
        i += 1
        if i>=stop_iter:
            break
```

After training, we get result like this:



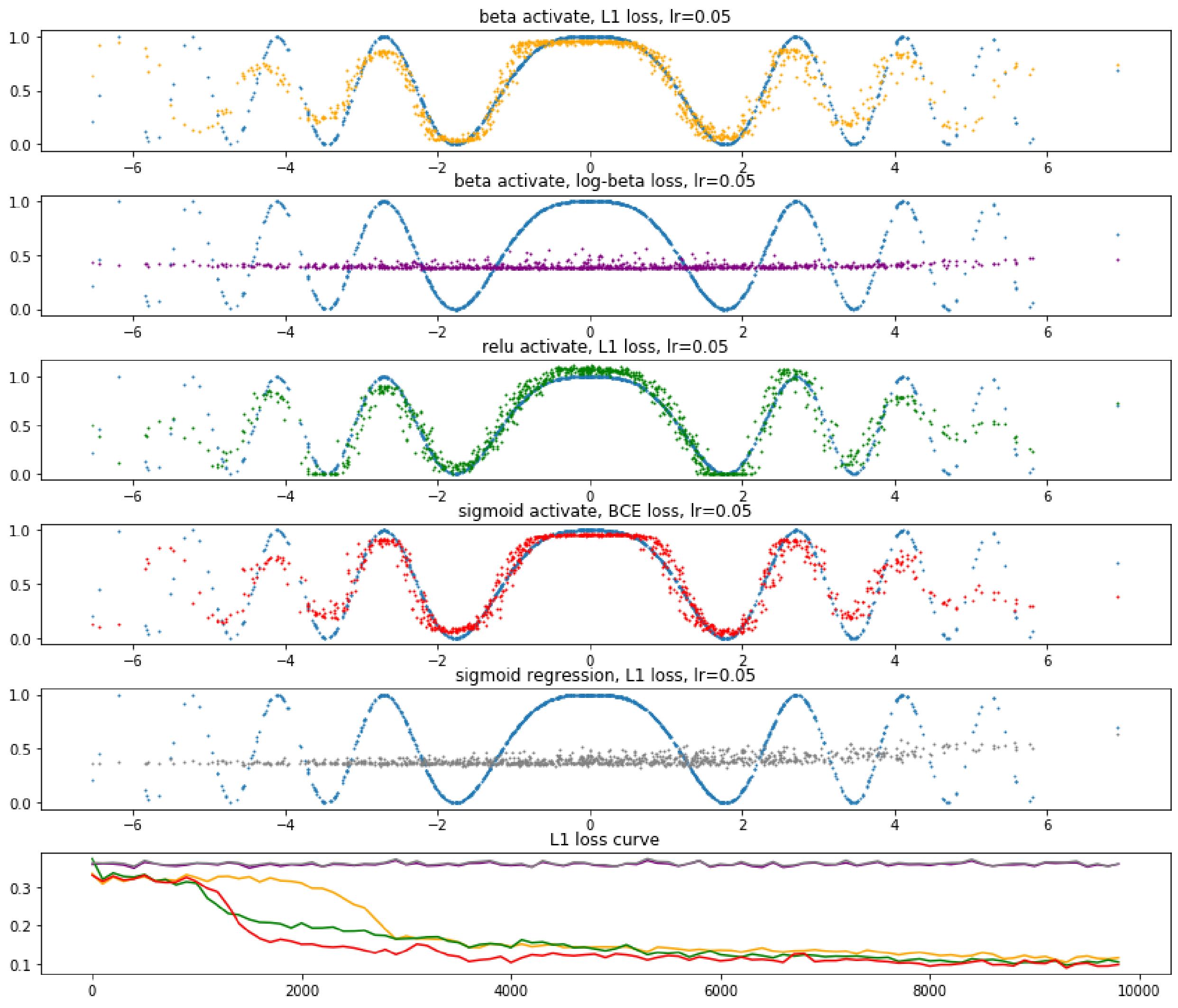
As we could see: the beta-activation is a feasible candidate in the bounded regression problem setting, although it did not demonstrate salient superiority compared with relu-activation and sigmoid-activation.

However, when using negative-beta-likelihood as loss function, I observed failure:

```
def beta_loss(y_hat, gamma , y_true):
    a = y_hat*gamma
    b = gamma - a
    tmp1 = (a-1)*torch.lgamma(y_true)
    tmp2 = (b-1)*torch.lgamma(y_true)
    tmp3 = torch.lgamma(a+b)-torch.lgamma(a)-torch.lgamma(b)
    return torch.exp(-torch.sum(tmp1+tmp2+tmp3)*0.01)

while 1:
    x, y_true = next(train_iter)

    beta_nn_ml.train()
    y_hat1, gamma1 = beta_nn_ml(x)
    loss1 = beta_loss(y_hat1, gamma1 , y_true)
```

```
        loss1.backward()
        loss_buf1 += F.l1_loss(y_hat1, y_true).detach()
        optimizer1.step()
        optimizer1.zero_grad()

        # hidden: print and update loss buffer
        i += 1
        if i>=stop_iter:
            break
```

The result comes:



Therefore, future problems are to:

1. find typical user cases for beta-activate;
2. design a suitable loss function for beta-activated NN

The code for this test is here (in the notebook):

BoundedRegression.ipynb
227.6 KB