lab-5-logistic-regression-cross-entropy-one-hot-encoding

March 4, 2023

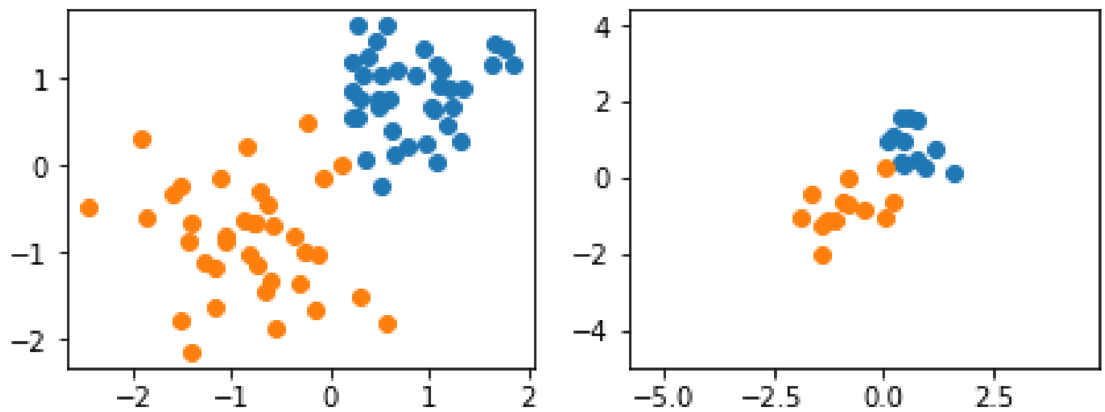
[2]:
```
import numpy as np
import pandas as pd
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

0.1 Logistic Regression

[6]:
```
data = np.genfromtxt('data/logistic-toydata.txt', delimiter='\t')
x = data[:, :2].astype(np.float32)
y = data[:, 2].astype(np.int64)

np.random.seed(123)
idx = np.arange(y.shape[0])
np.random.shuffle(idx)
X_test, y_test = x[idx[:25]], y[idx[:25]]
X_train, y_train = x[idx[25:]], y[idx[25:]]
mu, std = np.mean(X_train, axis=0), np.std(X_train, axis=0)
X_train, X_test = (X_train - mu) / std, (X_test - mu) / std

fig, ax = plt.subplots(1, 2, figsize=(7, 2.5))
ax[0].scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1])
ax[0].scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1])
ax[1].scatter(X_test[y_test == 1, 0], X_test[y_test == 1, 1])
ax[1].scatter(X_test[y_test == 0, 0], X_test[y_test == 0, 1])
plt.xlim([x[:, 0].min()-0.5, x[:, 0].max()+0.5])
plt.ylim([x[:, 1].min()-0.5, x[:, 1].max()+0.5])
plt.show()
```



0.1.1 Implementation with manual gradient

Gradient descent rule for Logistic Regression

$$z = \mathbf{w}^T\mathbf{x} + b$$

$$a = \sigma(z)$$

The idea of the loss function in the logistic regression is such that we take $\sigma(z)$ if true $y = 1$ and $1 - \sigma(z)$ if true $y = 0$. We can incoporate this information in a compact notation as follows:

We want to maximize the probability along all the data points, i.e. the loss function $L(\mathbf{w})$ needs to be maximised

$$L(\mathbf{w}) = \prod_{i=1}^{n} P(y^{[i]}|x^{[i]}; \mathbf{w})$$

$$L(\mathbf{w}) = \prod_{i=1}^{n} ((\sigma(z^i))^{y^i})((1 - \sigma(z^i))^{1-y^i})$$

Lets introduce loss and call it *log-likelihood*, so we write the above equation as

$$l(\mathbf{w}) = log(L(\mathbf{w}))$$

which equals

In practise we will *minimize negative log-likelihood* instead of maximizing log-likelihood. So,

$$l(\mathbf{w}) = -log(L(\mathbf{w}))$$

should be minimized

$$\frac{\delta L}{\delta w} = \frac{\delta L}{\delta a} * \frac{\delta a}{\delta z} * \frac{\delta z}{\delta w}$$

Long story short:

$$\frac{\delta L}{\delta a} = \frac{a-y}{a-a^2}$$
$$\frac{\delta a}{\delta z} = a.(1-a)$$
$$\frac{\delta a}{\delta w} = x$$

$$\frac{\delta L}{\delta z} = \frac{\delta L}{\delta a}\frac{\delta a}{\delta z} = \frac{a-y}{a-a^2}a.(1-a) = a-y$$

$$\frac{\delta L}{\delta w} = \frac{\delta L}{\delta a}\frac{\delta a}{\delta z}\frac{\delta z}{\delta w} = (a-y)x$$

[30]:
```
device  = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class LogisticRegression1():
    def __init__(self, num_features):
        self.num_features = num_features;
        self.weight = torch.zeros(1,num_features, dtype=torch.float32,␣
 ↪device=device)
        self.bias = torch.zeros(1, dtype=torch.float, device=device)

    def forward(self, x):
        linear = torch.add(torch.mm(x, self.weight.t()), self.bias).view(-1)
        probas = self._sigmoid(linear)
        return probas;

    def backward(self, x,y,probas):
        grad_loss_wrt_z = probas.view(-1) - y #dl/dz = (dl/da)*(da/dz) = a-y
        grad_loss_wrt_w = torch.mm(x.t(), grad_loss_wrt_z.view(-1, 1)).t()
        grad_loss_wrt_b = torch.sum(grad_loss_wrt_z)
        return grad_loss_wrt_w, grad_loss_wrt_b;

    def predict_labels(self,x):
        probas = self.forward(x)
        labels = torch.where(probas >= .5, 1, 0) # threshold function
        return labels;

    def evaluate(self, x, y):
        labels = self.predict_labels(x).float()
        accuracy = torch.sum(labels.view(-1) == y.float()).item() / y.size(0)
        return accuracy
```

```
    def _sigmoid(self, z):
        return 1. / (1. + torch.exp(-z));

    def _logit_cost(self, y, proba):
        tmp1 = torch.mm(-y.view(1, -1), torch.log(proba.view(-1, 1)))
        tmp2 = torch.mm((1 - y).view(1, -1), torch.log(1 - proba.view(-1, 1)))
        return tmp1 - tmp2

    def train(self, x, y, num_epochs, learning_rate=0.01):
        epoch_cost = []
        for e in range(num_epochs):

            #### Compute outputs ####
            probas = self.forward(x)

            #### Compute gradients ####
            grad_w, grad_b = self.backward(x, y, probas)

            #### Update weights ####
            self.weight -= learning_rate * grad_w
            self.bias -= learning_rate * grad_b

            #### Logging ####
            cost = self._logit_cost(y, self.forward(x)) / x.size(0)
            print('Epoch: %03d' % (e+1), end="")
            print(' | Train ACC: %.3f' % self.evaluate(x, y), end="")
            print(' | Cost: %.3f' % cost)
            epoch_cost.append(cost)
        return epoch_cost
```

[32]:

```
X_train_tensor = torch.tensor(X_train, dtype=torch.float32, device=device)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32, device=device)

model1 = LogisticRegression1(num_features=2)
epoch_cost = model1.train(X_train_tensor, y_train_tensor, num_epochs=30,
 ↪learning_rate=0.1)

print('\nModel parameters:')
print('  Weights: %s' % model1.weight)
print('  Bias: %s' % model1.bias)
```

```
Epoch: 001 | Train ACC: 0.973 | Cost: 0.055
Epoch: 002 | Train ACC: 0.973 | Cost: 0.053
Epoch: 003 | Train ACC: 0.973 | Cost: 0.051
Epoch: 004 | Train ACC: 0.973 | Cost: 0.049
Epoch: 005 | Train ACC: 0.973 | Cost: 0.048
```

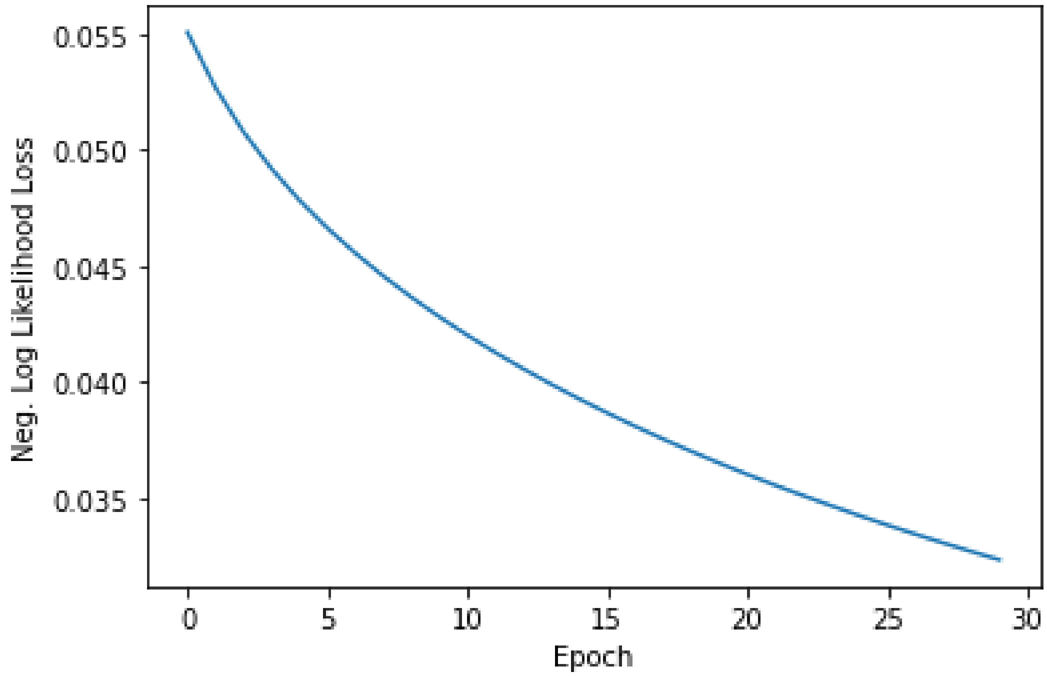
```
Epoch: 006 | Train ACC: 0.973 | Cost: 0.047
Epoch: 007 | Train ACC: 0.973 | Cost: 0.046
Epoch: 008 | Train ACC: 0.973 | Cost: 0.045
Epoch: 009 | Train ACC: 0.973 | Cost: 0.044
Epoch: 010 | Train ACC: 0.987 | Cost: 0.043
Epoch: 011 | Train ACC: 0.987 | Cost: 0.042
Epoch: 012 | Train ACC: 0.987 | Cost: 0.041
Epoch: 013 | Train ACC: 0.987 | Cost: 0.041
Epoch: 014 | Train ACC: 0.987 | Cost: 0.040
Epoch: 015 | Train ACC: 0.987 | Cost: 0.039
Epoch: 016 | Train ACC: 0.987 | Cost: 0.039
Epoch: 017 | Train ACC: 1.000 | Cost: 0.038
Epoch: 018 | Train ACC: 1.000 | Cost: 0.038
Epoch: 019 | Train ACC: 1.000 | Cost: 0.037
Epoch: 020 | Train ACC: 1.000 | Cost: 0.036
Epoch: 021 | Train ACC: 1.000 | Cost: 0.036
Epoch: 022 | Train ACC: 1.000 | Cost: 0.036
Epoch: 023 | Train ACC: 1.000 | Cost: 0.035
Epoch: 024 | Train ACC: 1.000 | Cost: 0.035
Epoch: 025 | Train ACC: 1.000 | Cost: 0.034
Epoch: 026 | Train ACC: 1.000 | Cost: 0.034
Epoch: 027 | Train ACC: 1.000 | Cost: 0.033
Epoch: 028 | Train ACC: 1.000 | Cost: 0.033
Epoch: 029 | Train ACC: 1.000 | Cost: 0.033
Epoch: 030 | Train ACC: 1.000 | Cost: 0.032

Model parameters:
  Weights: tensor([[5.0453, 3.4349]])
  Bias: tensor([-0.7931])
```

[33]:
```python
plt.plot(epoch_cost)
plt.ylabel('Neg. Log Likelihood Loss')
plt.xlabel('Epoch')
plt.show()
```

```
/opt/anaconda3/lib/python3.9/site-packages/numpy/core/shape_base.py:65:
FutureWarning: The input object of type 'Tensor' is an array-like implementing
one of the corresponding protocols (`__array__`, `__array_interface__` or
`__array_struct__`); but not a sequence (or 0-D). In the future, this object
will be coerced as if it was first converted using `np.array(obj)`. To retain
the old behaviour, you have to either modify the type 'Tensor', or assign to an
empty array created with `np.empty(correct_shape, dtype=object)`.
  ary = asanyarray(ary)
/opt/anaconda3/lib/python3.9/site-packages/numpy/core/shape_base.py:65:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray.
```

```
ary = asanyarray(ary)
```



0.1.2 Implementation with Pytorch nn.Module API

[40]:
```
device  = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
class LogisticRegression2(torch.nn.Module):
    def __init__(self, num_features):
        super(LogisticRegression2, self).__init__()
        self.linear = torch.nn.Linear(num_features, 1, dtype=torch.float32,␣
 ↪device = device)

        # initialize weights to zeros here,
        # since we used zero weights in the
        # manual approach

        self.linear.weight.detach().zero_()
        self.linear.bias.detach().zero_()

    def forward(self, x):
        logits = self.linear(x);
        probas = torch.sigmoid(logits)
        return probas;
model2 = LogisticRegression2(2).to(device)
optimizer = torch.optim.SGD(model2.parameters(), lr=0.1)
```

[42]:
```python
def comp_accuracy(label_var, pred_probas):
    pred_labels = torch.where((pred_probas > 0.5), 1, 0).view(-1)
    acc = torch.sum(pred_labels == label_var.view(-1)).float() / label_var.
 ↪size(0)
    return acc

num_epochs = 30

X_train_tensor = torch.tensor(X_train, dtype=torch.float32, device=device)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32, device=device).
 ↪view(-1, 1)


for epoch in range(num_epochs):

    #### Compute outputs ####
    out = model2(X_train_tensor)

    #### Compute gradients ####
    loss = F.binary_cross_entropy(out, y_train_tensor, reduction='sum')
    optimizer.zero_grad()
    loss.backward()

    #### Update weights ####
    optimizer.step()

    #### Logging ####
    pred_probas = model2(X_train_tensor)
    acc = comp_accuracy(y_train_tensor, pred_probas)
    print('Epoch: %03d' % (epoch + 1), end="")
    print(' | Train ACC: %.3f' % acc, end="")
    print(' | Cost: %.3f' % F.binary_cross_entropy(pred_probas, y_train_tensor))


print('\nModel parameters:')
print('  Weights: %s' % model2.linear.weight)
print('  Bias: %s' % model2.linear.bias)
```

```
Epoch: 001 | Train ACC: 0.973 | Cost: 0.055
Epoch: 002 | Train ACC: 0.973 | Cost: 0.053
Epoch: 003 | Train ACC: 0.973 | Cost: 0.051
Epoch: 004 | Train ACC: 0.973 | Cost: 0.049
Epoch: 005 | Train ACC: 0.973 | Cost: 0.048
Epoch: 006 | Train ACC: 0.973 | Cost: 0.047
Epoch: 007 | Train ACC: 0.973 | Cost: 0.046
Epoch: 008 | Train ACC: 0.973 | Cost: 0.045
Epoch: 009 | Train ACC: 0.973 | Cost: 0.044
```

```
Epoch: 010 | Train ACC: 0.987 | Cost: 0.043
Epoch: 011 | Train ACC: 0.987 | Cost: 0.042
Epoch: 012 | Train ACC: 0.987 | Cost: 0.041
Epoch: 013 | Train ACC: 0.987 | Cost: 0.041
Epoch: 014 | Train ACC: 0.987 | Cost: 0.040
Epoch: 015 | Train ACC: 0.987 | Cost: 0.039
Epoch: 016 | Train ACC: 0.987 | Cost: 0.039
Epoch: 017 | Train ACC: 1.000 | Cost: 0.038
Epoch: 018 | Train ACC: 1.000 | Cost: 0.038
Epoch: 019 | Train ACC: 1.000 | Cost: 0.037
Epoch: 020 | Train ACC: 1.000 | Cost: 0.036
Epoch: 021 | Train ACC: 1.000 | Cost: 0.036
Epoch: 022 | Train ACC: 1.000 | Cost: 0.036
Epoch: 023 | Train ACC: 1.000 | Cost: 0.035
Epoch: 024 | Train ACC: 1.000 | Cost: 0.035
Epoch: 025 | Train ACC: 1.000 | Cost: 0.034
Epoch: 026 | Train ACC: 1.000 | Cost: 0.034
Epoch: 027 | Train ACC: 1.000 | Cost: 0.033
Epoch: 028 | Train ACC: 1.000 | Cost: 0.033
Epoch: 029 | Train ACC: 1.000 | Cost: 0.033
Epoch: 030 | Train ACC: 1.000 | Cost: 0.032

Model parameters:
  Weights: Parameter containing:
tensor([[5.0453, 3.4349]], requires_grad=True)
  Bias: Parameter containing:
tensor([-0.7931], requires_grad=True)
```

[44]:
```python
X_test_tensor = torch.tensor(X_test, dtype=torch.float32, device=device)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32, device=device)

pred_probas = model2(X_test_tensor)
test_acc = comp_accuracy(y_test_tensor, pred_probas)

print('Test set accuracy: %.2f%%' % (test_acc*100))
```

```
Test set accuracy: 96.00%
```

0.1.3 Cross Entropy Example

One Hot Encoding

[45]:
```python
def to_onehot(y, num_classes):
    y_onehot = torch.zeros(y.size(0), num_classes)
    y_onehot.scatter_(1, y.view(-1, 1).long(), 1).float()
    return y_onehot

y = torch.tensor([0, 1, 2, 2])
```

```
y_enc = to_onehot(y, 3)

print('one-hot encoding:\n', y_enc)
```

```
one-hot encoding:
 tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.],
        [0., 0., 1.]])
```

Softmax Suppose we have some net inputs Z, where each row is one training example: Recall that the number of columns in **w** is equal to the number of labels for a classification setting. The number of rows corresponds to the number of samples in the dataset

[46]:
```
Z = torch.tensor( [[-0.3,  -0.5, -0.5],
                   [-0.4,  -0.1, -0.5],
                   [-0.3,  -0.94, -0.5],
                   [-0.99, -0.88, -0.5]])

Z
```

[46]:
```
tensor([[-0.3000, -0.5000, -0.5000],
        [-0.4000, -0.1000, -0.5000],
        [-0.3000, -0.9400, -0.5000],
        [-0.9900, -0.8800, -0.5000]])
```

The z output will input into ONE activation function. Recall that in two-class classification problem we had only one column of z which was input into the activation

Convert z to probabilies via softmax (j is the number of class labels):

$$P(y = j|z^i) = \sigma_{softmax}(z^i) = \frac{\exp^{z^{(i)}}}{\sum_{j=0}^{k} \exp^{z_k^{(i)}}}$$

Essentially, softmax is just an exponential function that normalises the activations so that they sum upto 1

[47]:
```
def softmax(z):
    return (torch.exp(z.t()) / torch.sum(torch.exp(z), dim=1)).t()

smax = softmax(Z)
print('softmax:\n', smax)
```

```
softmax:
 tensor([[0.3792, 0.3104, 0.3104],
        [0.3072, 0.4147, 0.2780],
        [0.4263, 0.2248, 0.3490],
        [0.2668, 0.2978, 0.4354]])
```

[50]:
```
torch.sum(smax, dim=1)
```

[50]:
```
tensor([1., 1., 1., 1.])
```

[51]:
```
def to_classlabel(z):
    return torch.argmax(z, dim=1)

print('predicted class labels: ', to_classlabel(smax))
print('true class labels: ', to_classlabel(y_enc))
```

```
predicted class labels:  tensor([0, 1, 0, 2])
true class labels:  tensor([0, 1, 2, 2])
```

Cross Entropy Now we will compute the cross entropy for each training example:

$$L(\mathbf{w}; \mathbf{b}) = \sum_{i=1}^{n} H(T_i, O_i)$$

$$H(T_i, O_i) = -\sum_{m} T_i.log(O_i)$$

[53]:
```
def cross_entropy(softmax, y_target):
    return - torch.sum(torch.log(softmax) * (y_target), dim=1) #dim=1 ensures␣
 ↪that the sum is taken for each row (or traning examples)

xent = cross_entropy(smax, y_enc)
print('Cross Entropy:', xent)
```

```
Cross Entropy: tensor([0.9698, 0.8801, 1.0527, 0.8314])
```

In Pytorch

[55]:
```
import torch.nn.functional as F
```

[58]:
```
# Note that nll_loss takes log(softmax) as input:
F.nll_loss(torch.log(smax), y, reduction='none')
```

[58]:
```
tensor([0.9698, 0.8801, 1.0527, 0.8314])
```

[59]:
```
# Note that cross_entropy takes logits as input:
F.cross_entropy(Z, y, reduction='none')
```

[59]:
```
tensor([0.9698, 0.8801, 1.0527, 0.8314])
```

[60]:
```
# if you dont use reduction='none', then the average is returned
F.cross_entropy(Z, y)
```

[60]: tensor(0.9335)

[ ]:

```

```