homework3

February 12, 2019

1 Homework 3 - Berkeley STAT 157

Handout 2/5/2019, due 2/12/2019 by 4pm in Git by committing to your repository.

Formatting: please include both a .ipynb and .pdf file in your homework submission, named homework3.ipynb and homework3.pdf. You can export your notebook to a pdf either by File -> Download as -> PDF via Latex (you may need Latex installed), or by simply printing to a pdf from your browser (you may want to do File -> Print Preview in jupyter first). Please don't change the filename.

```
In [1]: from mxnet import nd, autograd, gluon
        import matplotlib.pyplot as plt
```

2 1. Logistic Regression for Binary Classification

In multiclass classification we typically use the exponential model

$$p(y|\mathbf{o}) = \mathrm{softmax}(\mathbf{o})_y = \frac{\exp(o_y)}{\sum_{y'} \exp(o_{y'})}$$

1.1. Show that this parametrization has a spurious degree of freedom. That is, show that both $\mathbf{o}$ and $\mathbf{o} + c$ with $c \in \mathbb{R}$ lead to the same probability estimate. 1.2. For binary classification, i.e. whenever we have only two classes $\{-1, 1\}$, we can arbitrarily set $o_{-1} = 0$. Using the shorthand $o = o_1$ show that this is equivalent to

$$p(y = 1|o) = \frac{1}{1 + \exp(-o)}$$

1.3. Show that the log-likelihood loss (often called logistic loss) for labels $y \in \{-1, 1\}$ is thus given by

$$-\log p(y|o) = \log(1 + \exp(-y \cdot o))$$

1.4. Show that for $y = 1$ the logistic loss asymptotes to o for $o \to \infty$ and to $\exp(o)$ for $o \to -\infty$.

2.0.1 1.1

$$\frac{\exp(o_y)}{\sum_{y'} \exp(o_{y'})}$$

$$= \frac{\exp(o_y + c)}{\sum_{y'} \exp(o_{y'+c})}$$

$$= \frac{\exp(x + c)}{\sum_i \exp(x_i + c)}$$

$$= \frac{\exp(x) \cdot \exp(c)}{\sum \exp(x_i) \cdot \exp(c)}$$

$$= \frac{\exp(x) \cdot \exp(c)}{n \exp(c) \sum_i \exp(x_i)}$$

$$= \frac{\exp(x)}{n \sum \exp(x_i)}$$

$$= \frac{1}{n} \frac{\exp(x)}{\sum \exp(x_i)} = \frac{1}{n} \frac{\exp(o_y)}{\sum_{y'} \exp(o_{y'})} = \frac{1}{n} softmax(\mathbf{o}_y)$$

The last statement is the same as the first statement but the result is just rescaled by $1/n$ where n is the number of distinct labels.

1.2

$$\frac{exp(o_1)}{exp(o_1) + exp(o_{-1})} = \frac{1}{1 + \exp(-o_1))}$$

where $o_{-1} = 0$

$$\frac{exp(o)}{exp(o) + 1} = \frac{1}{1 + \exp(-o))}$$

$$\exp(o)(1 + \exp(-o)) = 1 + \exp(o)$$

$$\exp(o) + \exp(-o) \exp(o) = 1 + \exp(o)$$

$$\exp(o) + \exp(-o + o) = 1 + \exp(o)$$

$$\exp(o) + 1 \ = 1 + \exp(o)$$

1.3

$$-\log \frac{1}{1 + e^{-o}} = \log(1 + e^{-y \cdot o})$$

$$= -(\log 1 - \log(1 + e^{-o})) = \log(1 + e^{-y \cdot o})$$

$$= \log(1 + e^{-o}) - \log 1 = \log(1 + e^{-y \cdot o})$$

$$= \log(1 + e^{-o}) = \log(1 + e^{-y \cdot o})$$

$$= 1 + e^{-o} = 1 + e^{-y \cdot o}$$

Since it is binary and as mentioned in part 1.2, we can arbitrarily set o_{-1} to 0. So if $y = 0$, then the equation becomes 0 on both sides. If it is 1, they are $e^{-o} = e^{-o}$

1.4

$$\log(1+e^{-x}) \to 0$$

when $x \to \infty$ and

$$\log(1+e^{-x}) \to e^{x}$$

when $x \to -\infty$

1. $\log(1+e^{-x}) \to \log(\frac{e^x+1}{e^x}) \to \log(e^x+1) - \log(e^x)$. As $x \to \infty$, both equation goes to infinity and their difference goes to 0 but since the first equation is always 1 greater than the second, it doesn't actually reach 0.
2. $\log(1+\frac{1}{e^x})$. The e^x goes to ∞ as $x \to \infty$ but since it is now inversed, it goes to zero. However since our x goes to $-\infty$, this effect also becomes inverse that goes to the left side of the graph that it acts like a regular e^x that as $x \to -\infty$, the equation becomes e^x.

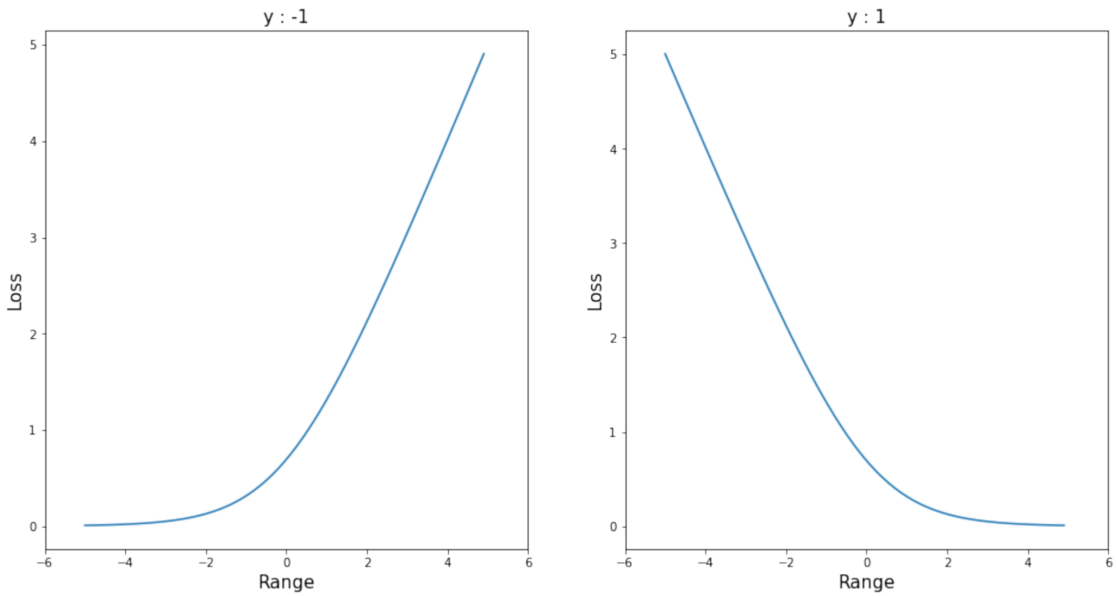
3 2. Logistic Regression and Autograd

1. Implement the binary logistic loss $l(y, o) = \log(1 + \exp(-y \cdot o))$ in Gluon
2. Plot its values for $y \in \{-1, 1\}$ over the range of $o \in [-5, 5]$.
3. Plot its derivative with respect to o for $o \in [-5, 5]$ using 'autograd'.

In [2]:
```
loss = gluon.loss.LogisticLoss()
```

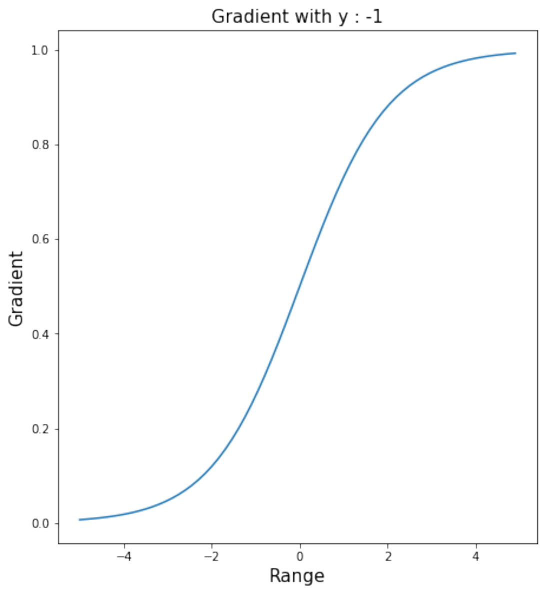
In [3]:
```
pred = nd.arange(-5, 5, 0.15)

label = [-1 * nd.ones_like(pred), nd.ones_like(pred)]

l = [loss(pred, label[0]), loss(pred, label[1])]
```

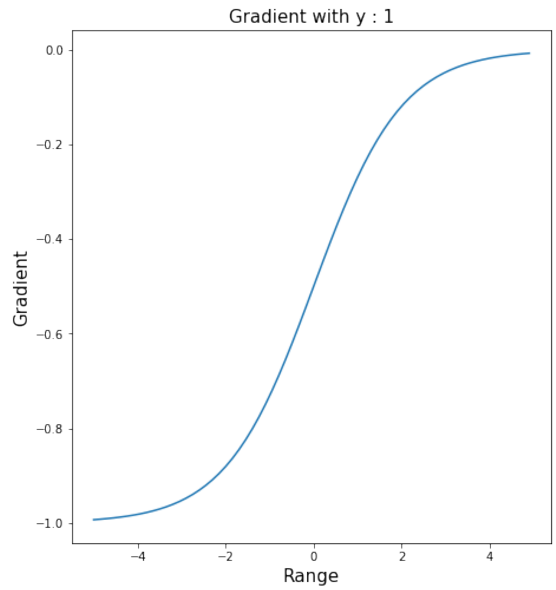
In [4]:
```
fig, ax = plt.subplots(1,2, figsize=(16, 8))
y = [-1, 1]

for i in range(2):

    ax[i].plot(pred.asnumpy(), l[i].asnumpy())
    ax[i].set_title(f'y : {y[i]}', size=15)
    ax[i].set_xlim(-6, 6)
    ax[i].set_ylabel('Loss',size=15)
    ax[i].set_xlabel('Range', size=15)

plt.show();
```




```
In [5]: pred1 = pred.copy()
        pred2 = pred.copy()

        pred1.attach_grad()
        pred2.attach_grad()

        with autograd.record():

            do1 = loss(pred1, label[0])
            do2 = loss(pred2, label[1])

        do1.backward()
        do2.backward()

        dpred = [pred1, pred2]
```

```
In [6]: fig, ax = plt.subplots(1,2, figsize=(16, 8))

        for i in range(2):

            ax[i].plot(pred.asnumpy(), dpred[i].grad.asnumpy())
            ax[i].set_title(f'Gradient with y : {y[i]}', size=15)
            ax[i].set_ylabel('Gradient', size=15)
            ax[i].set_xlabel('Range', size=15)


        plt.show();
```





4 3. Ohm's Law

Imagine that you're a young physicist, maybe named Georg Simon Ohm, trying to figure out how current and voltage depend on each other for resistors. You have some idea but you aren't quite sure yet whether the dependence is linear or quadratic. So you take some measurements, conveniently given to you as 'ndarrays' in Python. They are indicated by 'current' and 'voltage'.

Your goal is to use least mean squares regression to identify the coefficients for the following three models using automatic differentiation and least mean squares regression. The three models are:

1. Quadratic model where voltage $= c + r \cdot \text{current} + q \cdot \text{current}^2$.
2. Linear model where voltage $= c + r \cdot \text{current}$.
3. Ohm's law where voltage $= r \cdot \text{current}$.

In [7]:
```
current = nd.array([1.5420291, 1.8935232, 2.1603365, 2.5381863, 2.893443, \
                    3.838855, 3.925425, 4.2233696, 4.235571, 4.273397, \
                    4.9332876, 6.4704757, 6.517571, 6.87826, 7.0009003, \
                    7.035741, 7.278681, 7.7561755, 9.121138, 9.728281])
voltage = nd.array([63.802246, 80.036026, 91.4903, 108.28776, 122.781975, \
                    161.36314, 166.50816, 176.16772, 180.29395, 179.09758, \
                    206.21027, 272.71857, 272.24033, 289.54745, 293.8488, \
                    295.2281, 306.62274, 327.93243, 383.16296, 408.65967])
```

In [8]:
```
current = current.reshape(-1, 1)
voltage = voltage.reshape(-1, 1)
```

In [9]:
```
def data_iter(batch_size, features, labels):
```

```
import random

num_examples = len(features)
indices = list(range(num_examples))
# The examples are read at random, in no particular order
random.shuffle(indices)
for i in range(0, num_examples, batch_size):
    j = nd.array(indices[i: min(i + batch_size, num_examples)])
    yield features.take(j), labels.take(j)
    # The take function will then return the corresponding element based
    # on the indices
```

In [10]:
```
def sgd(params, lr, batch_size):
    for param in params:
        if param is None:
            continue
        param[:] = param - lr * param.grad / batch_size
```

In [11]:
```
def mse(y, y_hat):
    return nd.mean(nd.square(y - y_hat))
```

In [12]:
```
def model(features, labels, net, w1, w2=None, b=None, lr=0.01, num_epochs=5, loss=mse

    w1.attach_grad()

    if b is not None:
        b.attach_grad()

    if w2 is not None:
        w2.attach_grad()

    for epoch in range(num_epochs):

        for X, y in data_iter(batch_size, features, labels):

            with autograd.record():

                l = loss(net(X, w1, w2, b), y)

            l.backward()

            sgd([w1, w2, b], lr, batch_size)

        train_l = loss(labels, net(features, w1, w2, b))

        if epoch % iter_ver == 0:
            print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))

    return w1, w2, b
```

Quadratic Model

```
In [13]: def quad(X, w1, w2, b):
             return X * w1 + (X**2) * w2 + b

In [14]: r, q, c = model(current, voltage, quad, nd.random.normal(), nd.random.normal(), b=nd.z

epoch 1, loss 17263.068359
epoch 1001, loss 1009.545898
epoch 2001, loss 287.549500
epoch 3001, loss 85.914734
epoch 4001, loss 29.276550
epoch 5001, loss 13.472377
epoch 6001, loss 9.005405
epoch 7001, loss 7.691024
epoch 8001, loss 7.258520
epoch 9001, loss 7.071026


In [15]: print(f"Optimal value r, q and c for Linear model are : {r.asscalar(), q.asscalar(), c

Optimal value r, q and c for Linear model are : (37.18543, 0.4063551, 12.544488)
```

Linear Model

```
In [16]: def linear(X, w, w2, b):
             return X * w + b

In [17]: r, _, c = model(current, voltage, linear, nd.random.normal(), b=nd.zeros(1), num_epoch

epoch 1, loss 258.695831
epoch 6, loss 11.315999
epoch 11, loss 7.353860
epoch 16, loss 6.537631
epoch 21, loss 4.719824
epoch 26, loss 3.916882
epoch 31, loss 4.198429
epoch 36, loss 2.654677
epoch 41, loss 2.345388
epoch 46, loss 3.174131
epoch 51, loss 3.027113
epoch 56, loss 2.004357
epoch 61, loss 1.577524
epoch 66, loss 2.271075
epoch 71, loss 1.395448
epoch 76, loss 1.953512
epoch 81, loss 1.403591
epoch 86, loss 1.731378
```

```
epoch 91, loss 1.310277
epoch 96, loss 2.026414
```

```
In [18]: print(f"Optimal value r and c for Linear model are : {r.asscalar(), c.asscalar()}")
```

```
Optimal value r and c for Linear model are : (41.839638, 1.059855)
```

Ohm's Law

```
In [19]: def ohm(X, w, w2, b):
             return X * w
```

```
In [20]: r, *_ = model(current, voltage, ohm, nd.random.normal(), num_epochs=50, lr=0.1)
```

```
epoch 1, loss 1.507786
epoch 6, loss 1.739575
epoch 11, loss 2.139585
epoch 16, loss 1.932953
epoch 21, loss 1.553277
epoch 26, loss 1.287606
epoch 31, loss 1.334121
epoch 36, loss 1.431311
epoch 41, loss 1.370204
epoch 46, loss 1.308531
```

```
In [21]: print(f"Optimal value r for Ohm's formula is : {r.asscalar()}")
```

```
Optimal value r for Ohm's formula is : 41.91114044189453
```

I would say that the relationship is rather linear than quadratic.

5 4. Entropy

Let's compute the *binary* entropy of a number of interesting data sources.

1. Assume that you're watching the output generated by a monkey at a typewriter. The monkey presses any of the 44 keys of the typewriter at random (you can assume that it has not discovered any special keys or the shift key yet). How many bits of randomness per character do you observe?
2. Unhappy with the monkey you replaced it by a drunk typesetter. It is able to generate words, albeit not coherently. Instead, it picks a random word out of a vocabulary of $2,000$ words. Moreover, assume that the average length of a word is 4.5 letters in English. How many bits of randomness do you observe now?
3. Still unhappy with the result you replace the typesetter by a high quality language model. These can obtain perplexity numbers as low as 20 points per character. The perplexity is defined as a length normalized probability, i.e.

$$\mathrm{PPL}(x) = [p(x)]^{1/\mathrm{length}(x)}$$

1. $$44 * \frac{1}{44} * (-\log_2 \frac{1}{44}) = 5.46 bits/char$$

2. $$2000 * \frac{1}{2000} * (-\log_2 \frac{1}{2000}) = 10.97 bits/word = 2.44 bits/char$$

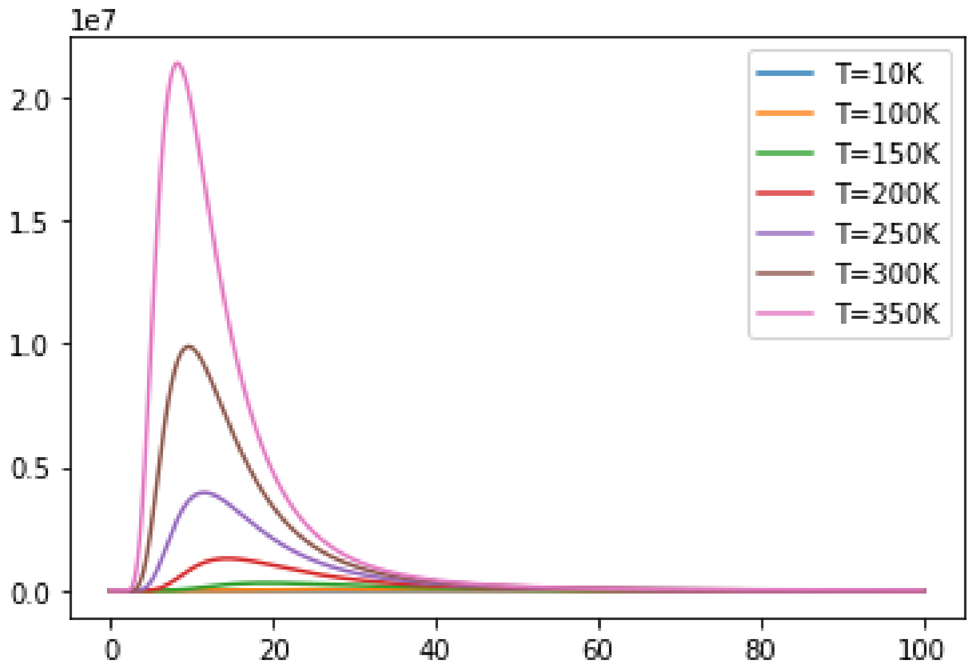
6 5. Wien's Approximation for the Temperature (bonus)

We will now abuse Gluon to estimate the temperature of a black body. The energy emanated from a black body is given by Wien's approximation.

$$B_\lambda(T) = \frac{2hc^2}{\lambda^5} \exp\left(-\frac{hc}{\lambda kT}\right)$$

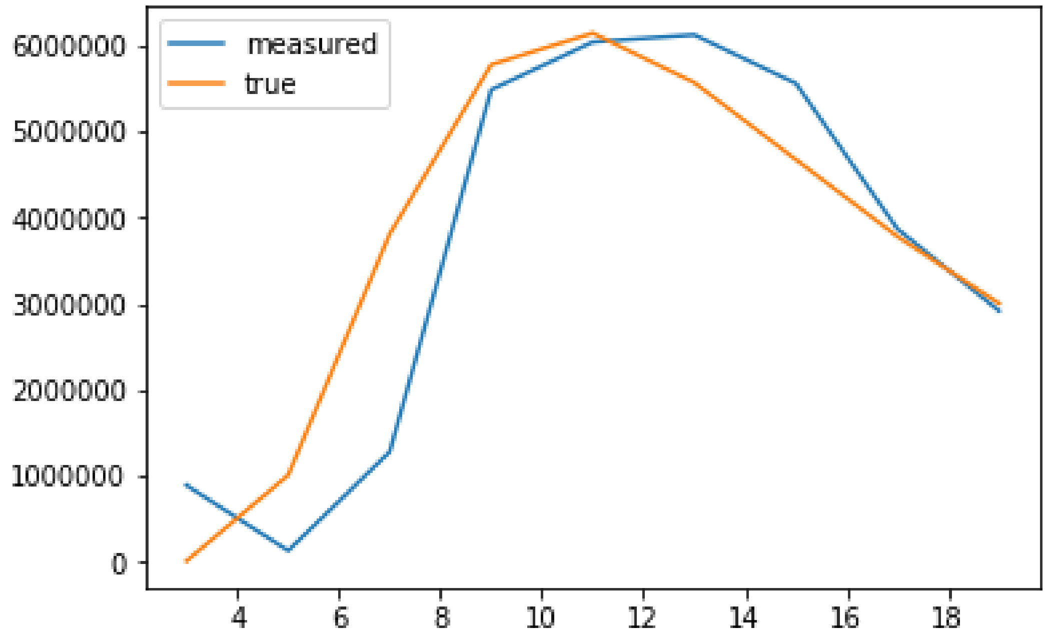
That is, the amount of energy depends on the fifth power of the wavelength λ and the temperature T of the body. The latter ensures a cutoff beyond a temperature-characteristic peak. Let us define this and plot it.

```
In [22]: # Lightspeed
         c = 299792458
         # Planck's constant
         h = 6.62607004e-34
         # Boltzmann constant
         k = 1.38064852e-23
         # Wavelength scale (nanometers)
         lamscale = 1e-6
         # Pulling out all powers of 10 upfront
         p_out = 2 * h * c**2 / lamscale**5
         p_in = (h / k) * (c/lamscale)

         # Wien's law
         def wien(lam, t):
             return (p_out / lam**5) * nd.exp(-p_in / (lam * t))

         # Plot the radiance for a few different temperatures
         lam = nd.arange(0,100,0.01)
         for t in [10, 100, 150, 200, 250, 300, 350]:
             radiance = wien(lam, t)
             plt.plot(lam.asnumpy(), radiance.asnumpy(), label=('T=' + str(t) + 'K'))
         plt.legend()
         plt.show()
```



Next we assume that we are a fearless physicist measuring some data. Of course, we need to pretend that we don't really know the temperature. But we measure the radiation at a few wavelengths.

```
In [23]: # real temperature is approximately 0C
         realtemp = 273
         # we observe at 3000nm up to 20,000nm wavelength
         wavelengths = nd.arange(3,20,2)
         # our infrared filters are pretty lousy ...
         delta = nd.random_normal(shape=(len(wavelengths))) * 1

         radiance = wien(wavelengths + delta,realtemp)
         plt.plot(wavelengths.asnumpy(), radiance.asnumpy(), label='measured')
         plt.plot(wavelengths.asnumpy(), wien(wavelengths, realtemp).asnumpy(), label='true')
         plt.legend()
         plt.show()
```



Use Gluon to estimate the real temperature based on the variables `wavelengths` and `radiance`.

- You can use Wien's law implementation `wien(lam,t)` as your forward model.
- Use the loss function $l(y, y') = (\log y - \log y')^2$ to measure accuracy.