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 [42]:
```
from mxnet import nd, autograd, gluon
from mxnet.gluon import data as gdata
from mxnet.gluon import nn
from mxnet import init
from mxnet.gluon import loss as gloss
import matplotlib.pyplot as plt
import numpy as np
```

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 $o \to -\infty$.

1.1

$$p(y|\mathbf{o} + \mathbf{c}) = \frac{\exp(o_y + c)}{\sum_{y'} \exp(o_{y'+c})} = \frac{\exp(o_y) * \exp(c)}{\exp(c) * \sum_{y'} \exp(o_{y'})} = \frac{\exp(o_y)}{\sum_{y'} \exp(o_{y'})} = p(y|\mathbf{o})$$

1.2

$$p(y=1|o) = \frac{\exp(o)}{\sum_{y'} \exp(o_{y'})} = \frac{\exp(o)}{\exp(o_{-1}) + \exp(o_1)} = \frac{\exp(o)}{\exp(0) + \exp(o_1)} = \frac{\exp(o)}{1 + \exp(o)} = \frac{\frac{\exp(o)}{\exp(o)}}{\frac{1}{\exp(o)} + \frac{\exp(}{\exp(}}$$

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

1.3

I am splitting it up into both y=1 and y=-1 since only options

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

now for y = -1

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

1.4

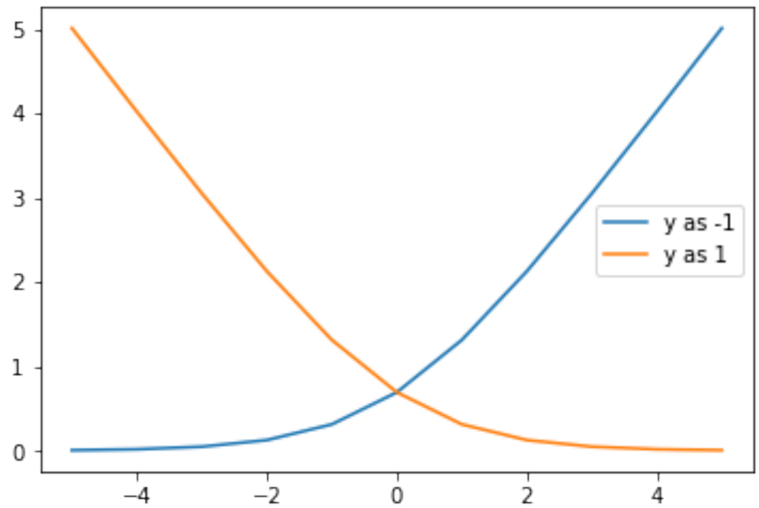
2. Logistic Regression and Autograd

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 [3]:
```
#1
def loss(y,o):
    ## add your loss function here
    return nd.log(1+nd.exp(-y*o))
```

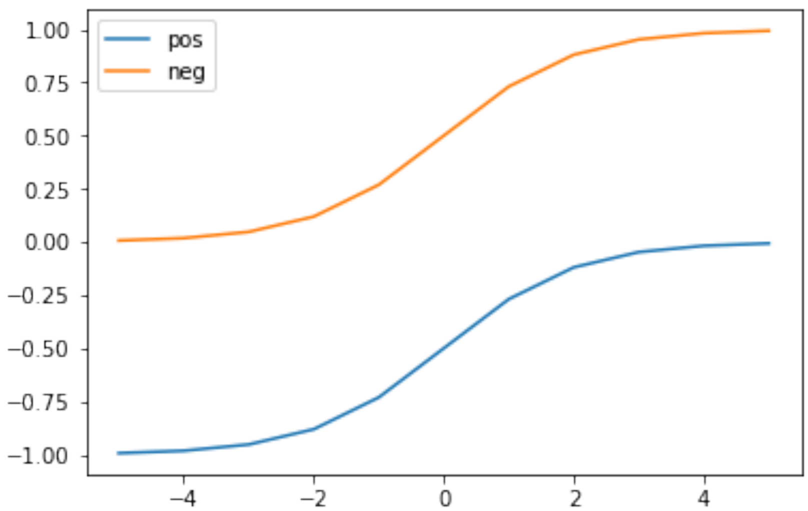
In [4]:
```
#2
o_vals = nd.arange(-5, 6)
plt.plot(o_vals.asnumpy(), loss(-1, o_vals).asnumpy(), label='y as -1'
)
plt.plot(o_vals.asnumpy(), loss(1, o_vals).asnumpy(), label= 'y as 1')
plt.legend()
```

Out[4]: <matplotlib.legend.Legend at 0x11d69d198>



In [5]:
```python
#Autograd derivative
o_for_pos = nd.arange(-5, 6)
o_for_neg = nd.arange(-5, 6)
o_for_pos.attach_grad()
o_for_neg.attach_grad()
with autograd.record():
    new_o_for_neg = loss(-1, o_for_neg)
    new_o_for_pos = loss(1, o_for_pos)
new_o_for_neg.backward()
new_o_for_pos.backward()
plt.plot(o_for_pos.asnumpy(), o_for_pos.grad.asnumpy(), label = 'pos')
plt.plot(o_for_neg.asnumpy(), o_for_neg.grad.asnumpy(), label = 'neg')
plt.legend()
```

Out[5]: <matplotlib.legend.Legend at 0x11d7167f0>



3. Ohm's Law

Imagine that you're a young physicist, maybe named [Georg Simon Ohm (https://en.wikipedia.org/wiki/Georg_Ohm)](https://en.wikipedia.org/wiki/Georg_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 $\text{voltage} = c + r \cdot \text{current} + q \cdot \text{current}^2$.
2. Linear model where $\text{voltage} = c + r \cdot \text{current}$.
3. Ohm's law where $\text{voltage} = r \cdot \text{current}$.

In [6]:
```
current = nd.array([1.5420291, 1.8935232, 2.1603365, 2.5381863, 2.8934
43, \
                    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, 122.7819
75, \
                    161.36314, 166.50816, 176.16772, 180.29395, 179.09
758, \
                    206.21027, 272.71857, 272.24033, 289.54745, 293.84
88, \
                    295.2281, 306.62274, 327.93243, 383.16296, 408.659
67])
```

In [64]:
```
current_sq = current**2
new_c = current.reshape(len(current), 1)
new_c_sq = current_sq.reshape(len(current), 1)
updated_current = nd.concat(new_c, new_c_sq)
updated_current
```

Out[64]:
```
[[ 1.5420291  2.3778539]
 [ 1.8935232  3.5854301]
 [ 2.1603365  4.6670537]
 [ 2.5381863  6.44239  ]
 [ 2.893443   8.372013 ]
 [ 3.838855  14.736808 ]
 [ 3.925425  15.408962 ]
 [ 4.2233696 17.836851 ]
 [ 4.235571  17.940062 ]
 [ 4.273397  18.26192  ]
 [ 4.9332876 24.337326 ]
 [ 6.4704757 41.867054 ]
 [ 6.517571  42.478733 ]
 [ 6.87826   47.310463 ]
 [ 7.0009003 49.012604 ]
 [ 7.035741  49.501648 ]
 [ 7.278681  52.979195 ]
 [ 7.7561755 60.15826  ]
 [ 9.121138  83.19515  ]
 [ 9.728281  94.63945  ]]
<NDArray 20x2 @cpu(0)>
```

In [77]:

```
#1
bs = 32

data_q1 = gdata.ArrayDataset(updated_current, voltage)
data_q1_iter = gdata.DataLoader(data_q1, bs, shuffle = True)

net_q1 = nn.Sequential()
net_q1.add(nn.Dense(1, use_bias = True))
net_q1.initialize(init.Normal(sigma=0.01))

loss_q1 = gloss.L2Loss()

trainer_q1 = gluon.Trainer(net_q1.collect_params(), 'sgd', {'learning_
rate':0.0001})

num_epoch = 25

for epoch in range(1, num_epoch + 1):
    for X, y in data_q1_iter:
        with autograd.record():
            l = loss_q1(net_q1(X), y)
        l.backward()
        trainer_q1.step(bs)
    l = loss_q1(net_q1(updated_current), voltage)
    print('epoch %d, loss: %f' %(epoch, l.mean().asnumpy()))

w_1 = net_q1[0].weight.data()
bias_1 = net_q1[0].bias.data()
print("weight and bias:", w_1, bias_1)
```

```
epoch 1, loss: 23351.488281
epoch 2, loss: 18845.189453
epoch 3, loss: 15289.289062
epoch 4, loss: 12483.317383
epoch 5, loss: 10269.085938
epoch 6, loss: 8521.777344
epoch 7, loss: 7142.899902
epoch 8, loss: 6054.739258
epoch 9, loss: 5195.972656
epoch 10, loss: 4518.213379
epoch 11, loss: 3983.279785
epoch 12, loss: 3561.044922
epoch 13, loss: 3227.738037
epoch 14, loss: 2964.599854
epoch 15, loss: 2756.829590
epoch 16, loss: 2592.748535
epoch 17, loss: 2463.140869
epoch 18, loss: 2360.735840
epoch 19, loss: 2279.794434
epoch 20, loss: 2215.789551
epoch 21, loss: 2165.148926
epoch 22, loss: 2125.053223
epoch 23, loss: 2093.278809
epoch 24, loss: 2068.069824
epoch 25, loss: 2048.041992
weight and bias:
[[0.8174263 5.144917 ]]
<NDArray 1x2 @cpu(0)>
[0.1546553]
<NDArray 1 @cpu(0)>
```

In [ ]:

```
```

In [ ]:

```
#2
```

In [79]:

```
batch_size = 20
data_q2 = gdata.ArrayDataset(current, voltage)
data_q2_iter = gdata.DataLoader(data_q2, batch_size, shuffle = True)
net_q2 = nn.Sequential()
net_q2.add(nn.Dense(1, use_bias = True))

net_q2.initialize(init.Normal(sigma=0.01))

loss_q2 = gloss.L2Loss()

trainer_q2 = gluon.Trainer(net_q2.collect_params(), 'sgd', {'learning_
rate':0.005})

num_epoch = 25

for epoch in range(1, num_epoch + 1):
    for X, y in data_q2_iter:
        with autograd.record():
            l2 = loss_q2(net_q2(X), y)
        l2.backward()
        trainer_q2.step(batch_size)
    l2 = loss_q2(net_q2(current), voltage)
    print('epoch %d, loss: %f' %(epoch, l2.mean().asnumpy()))

w_2 = net_q2[0].weight.data()
bias_2 = net_q2[0].bias.data()
print("weight and bias:", w_2, bias_2)
```

```
epoch 1, loss: 20086.898438
epoch 2, loss: 13907.583984
epoch 3, loss: 9629.552734
epoch 4, loss: 6667.807129
epoch 5, loss: 4617.344727
epoch 6, loss: 3197.777588
epoch 7, loss: 2214.989746
epoch 8, loss: 1534.589233
epoch 9, loss: 1063.536377
epoch 10, loss: 737.418274
epoch 11, loss: 511.640137
epoch 12, loss: 355.329132
epoch 13, loss: 247.111176
epoch 14, loss: 172.188751
epoch 15, loss: 120.317299
epoch 16, loss: 84.404480
epoch 17, loss: 59.539742
epoch 18, loss: 42.324104
epoch 19, loss: 30.404057
epoch 20, loss: 22.149780
epoch 21, loss: 16.433914
epoch 22, loss: 12.475170
epoch 23, loss: 9.732967
epoch 24, loss: 7.832946
epoch 25, loss: 6.516022
weight and bias:
[[40.613228]]
<NDArray 1x1 @cpu(0)>
[6.373268]
<NDArray 1 @cpu(0)>
```

In [25]:
```
#3
```

In [81]:

```python
batch_size = 20
data_q3 = gdata.ArrayDataset(current, voltage)
data_q3_iter = gdata.DataLoader(data_q3, batch_size, shuffle = True)
net_q3 = nn.Sequential()
net_q3.add(nn.Dense(1))

net_q3.initialize(init.Normal(sigma=0.01))

loss_q3 = gloss.L2Loss()

trainer_q3 = gluon.Trainer(net_q3.collect_params(), 'sgd', {'learning_
rate':0.01})

num_epoch = 25

for epoch in range(1, num_epoch + 1):
    for X, y in data_q3_iter:
        with autograd.record():
            l3 = loss_q3(net_q3(X), y)
        l3.backward()
        trainer_q3.step(batch_size)
    l3 = loss_q3(net_q3(current), voltage)
    print('epoch %d, loss: %f' %(epoch, l3.mean().asnumpy()))

w_3 = net_q3[0].weight.data()
bias_3 = net_q3[0].bias.data()
print("weight:", w_3)
```

```
epoch 1, loss: 12781.830078
epoch 2, loss: 5639.353027
epoch 3, loss: 2489.224121
epoch 4, loss: 1099.882568
epoch 5, loss: 487.120026
epoch 6, loss: 216.861664
epoch 7, loss: 97.660744
epoch 8, loss: 45.082996
epoch 9, loss: 21.888325
epoch 10, loss: 11.653015
epoch 11, loss: 7.133183
epoch 12, loss: 5.134256
epoch 13, loss: 4.247097
epoch 14, loss: 3.850332
epoch 15, loss: 3.669878
epoch 16, loss: 3.584819
epoch 17, loss: 3.541843
epoch 18, loss: 3.517469
epoch 19, loss: 3.501298
epoch 20, loss: 3.488767
epoch 21, loss: 3.477887
epoch 22, loss: 3.467721
epoch 23, loss: 3.457901
epoch 24, loss: 3.448235
epoch 25, loss: 3.438677
weight:
[[41.04584]]
<NDArray 1x1 @cpu(0)>
```

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 (https://en.wikipedia.org/wiki/File:Chimpanzee_seated_at_typewriter.jpg)](https://en.wikipedia.org/wiki/File:Chimpanzee_seated_at_typewriter.jpg). 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.

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

In [39]:
```
#H(X)=−∑i=1nPr[xi]⋅log2(Pr[xi])
val = 0
for i in nd.ones(44)/44:
    val += i * nd.log2(i)
val *= -1
print(val)
```

```
[5.45943]
<NDArray 1 @cpu(0)>
```

In [45]:
```
val_q2 = 0
for i in nd.ones(2000)/2000:
    val_q2 += i * nd.log2(i)
val_q2 *= -1 * np.log2(4.5)
print(val_q2)
```

```
[23.794353]
<NDArray 1 @cpu(0)>
```

In [47]:
```
val_q3 = 0
px = (nd.ones(2000)/2000)**1/2000
for i in px:
    val_q3 += i * nd.log2(i)
val_q3 *= -1
print(val_q3)
```

```
[0.01096555]
<NDArray 1 @cpu(0)>
```

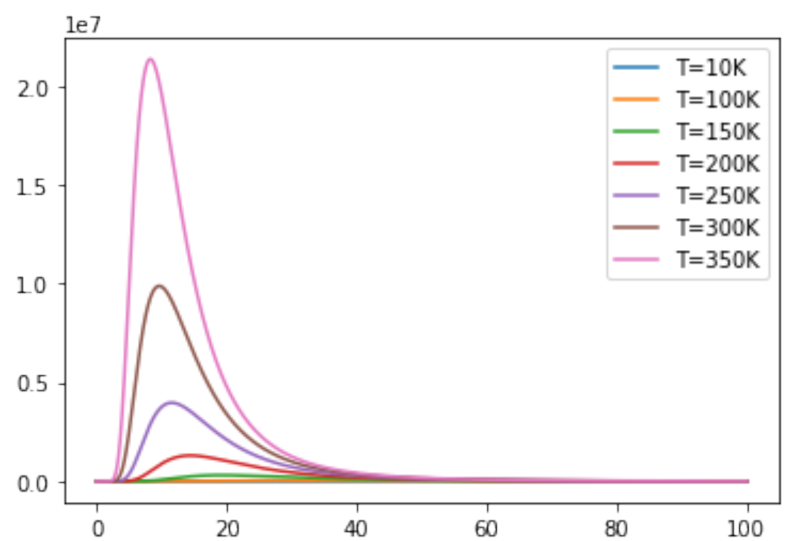
5. Wien's Approximation for the Temperature (bonus)

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.

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

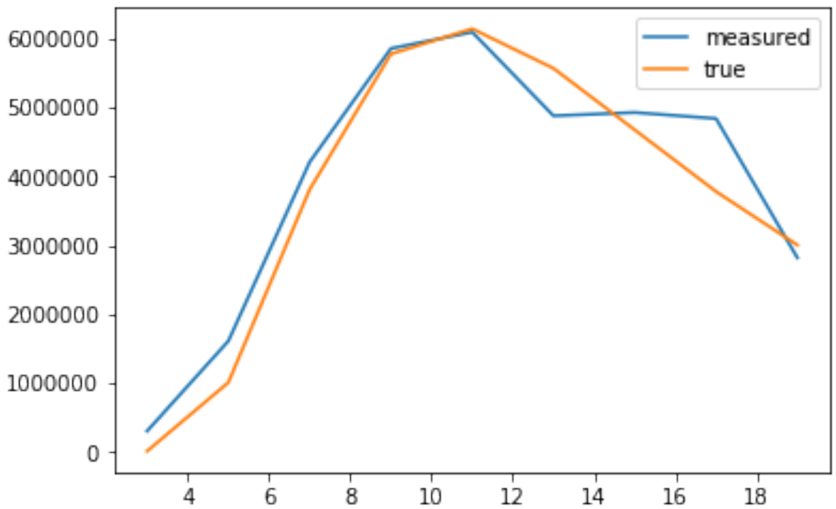
That is, the amount of energy depends on the fifth power of the wavelength λ and the temperature T of the body. The latter ensures a cutoff beyond a temperature-characteristic peak. Let us define this and plot it.

In [3]:
```
# 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 [4]:
```python
# 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.

In [ ]:
```python
batch_size = 10
data_set = yikes
```