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

1. Logistic Regression for Binary Classification

In multiclass classification we typically use the exponential model

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

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)

$$\text{softmax}(\mathbf{o} + c)_y =$$

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

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

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

1.2)

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

1.3)

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

and

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

means that

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

concluding that

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

1.4)

$$\lim_{o\to\infty} \log(1+\exp(-o)) \approx \log(1+\frac{1}{e^{\infty}}) = 0$$
$$\lim_{o\to\infty} \log(1+\exp(-o)) \approx \log(e^{o}) = o \text{ aka } \infty$$

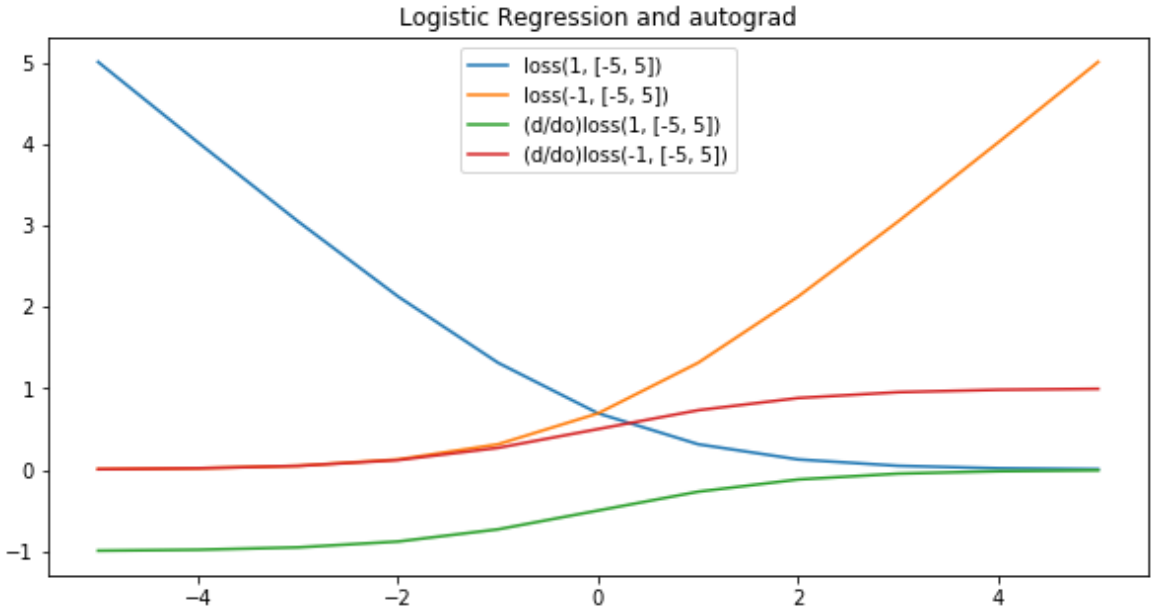
2. Logistic Regression and Autograd

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]:
```
def loss(y,o):
    l = nd.log(1+nd.exp(-y * o))
    return l
```

In [3]:
```
def dloss(y, o):
    o.attach_grad()
    with autograd.record():
        z = loss(y, o)
    z.backward()
    return o.grad
```

In [4]:
```
o = nd.arange(-5, 6)
plt.figure(figsize=(10,5))
plt.plot(o.asnumpy(), loss(1, o).asnumpy())
plt.plot(o.asnumpy(), loss(-1, o).asnumpy())
plt.plot(o.asnumpy(), dloss(1, o).asnumpy())
plt.plot(o.asnumpy(), dloss(-1, o).asnumpy())
plt.legend(['loss(1, [-5, 5])','loss(-1, [-5, 5])','(d/do)loss(1, [-
5, 5])','(d/do)loss(-1, [-5, 5])'])
plt.title("Logistic Regression and autograd")
plt.show()
```



3. Ohm's Law

Imagine that you're a young physicist, maybe named Georg Simon 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:

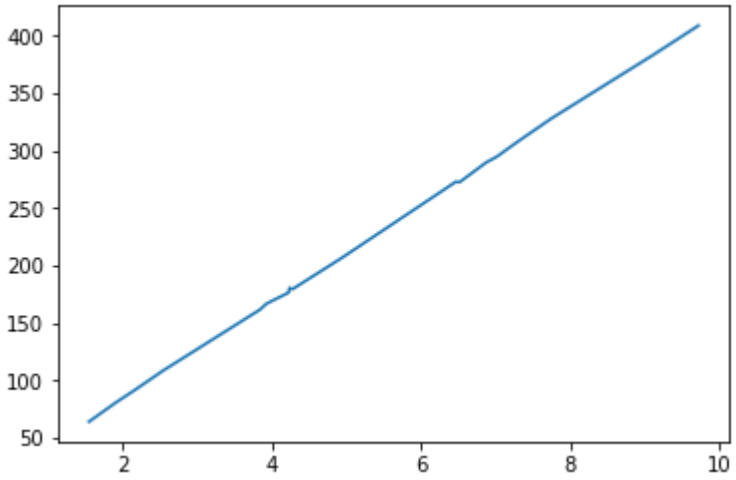
1. Quadratic model where $\text{voltage} = c + r \cdot \text{current} + q \cdot \text{current}^2$.
2. Linear model where $\text{voltage} = c + r \cdot \text{current}$.
3. Ohm's law where $\text{voltage} = r \cdot \text{current}$.

In [5]:
```
current = nd.array([1.5420291, 1.8935232, 2.1603365, 2.5381863, 2.893
443, \
                    3.838855, 3.925425, 4.2233696, 4.235571, 4.273397
, \
                    4.9332876, 6.4704757, 6.517571, 6.87826, 7.000900
3, \
                    7.035741, 7.278681, 7.7561755, 9.121138, 9.728281
])
voltage = nd.array([63.802246, 80.036026, 91.4903, 108.28776, 122.781
975, \
                    161.36314, 166.50816, 176.16772, 180.29395, 179.0
9758, \
                    206.21027, 272.71857, 272.24033, 289.54745, 293.8
488, \
                    295.2281, 306.62274, 327.93243, 383.16296, 408.65
967])
```

In [6]:
```
#plt.plot(current.asnumpy(), voltage.asnumpy())
```

In [7]:
```
plt.plot(current.asnumpy(), voltage.asnumpy())
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x7f3e9d0fdcf8>]
```



In [8]:
```
import random
def ohms(X, r):
    return nd.dot(X, r)
def linear(X, r, c):
    return nd.dot(X, r) + c
def quadratic(X, r, q, c):
    #print(nd.square(X))
    return nd.dot(nd.square(X), q) + nd.dot(nd.square(X), r) + c
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

In [9]:
```
r = nd.random.normal(scale=0.01, shape=(1, 1))
r.attach_grad()

lr = 0.03
num_epochs = 100
net = ohms
loss = squared_loss
for epoch in range(num_epochs):
    with autograd.record():
        for i in range(len(current)):
            l = loss(net(current[i], r), voltage[i])
    l.backward()
    sgd([r], lr, len(current))
print("loss", l)
print("Ohm's Law:", r[0].asscalar(), "* current = voltage")
```

```
loss
[7.450581e-09]
<NDArray 1 @cpu(0)>
Ohm's Law: 42.007374 * current = voltage
```

In [10]:
```
r = nd.random.normal(scale=0.01, shape=(1, 1))
c = nd.zeros(shape=(1,))
r.attach_grad()
c.attach_grad()

lr = 0.03
num_epochs = 100
net = linear
loss = squared_loss
for epoch in range(num_epochs):
    with autograd.record():
        for i in range(len(current)):
            l = loss(net(current[i], r, c), voltage[i])
    l.backward()
    sgd([r, c], lr, len(current))
print("loss", l)
print("Linear:", c[0].asscalar(), "+", r[0].asscalar(), "* current =
 voltage")
```

```
loss
[7.450581e-09]
<NDArray 1 @cpu(0)>
Linear: 4.2721305 + 41.56823 * current = voltage
```

In [11]:
```
r = nd.random.normal(scale=0.01, shape=(1, 1))
q = nd.random.normal(scale=0.01, shape=(1, 1))
c = nd.zeros(shape=(1,))
r.attach_grad()
c.attach_grad()
q.attach_grad()

lr = 0.03
num_epochs = 3
net = quadratic
loss = squared_loss
for epoch in range(num_epochs):
    with autograd.record():
        for i in range(len(current)):
            l = loss(net(current[i], r, q, c), voltage[i])
    l.backward()
    sgd([r, q, c], lr, len(current))
print("loss", l)
print("quadratic:", c[0].asscalar(), "+", r[0].asscalar(), "* current
+",  q[0].asscalar(), "* current^2","= voltage")
```

```
loss
[3.702368e+10]
<NDArray 1 @cpu(0)>
quadratic: 393.00742 + 37194.016 * current + 37194.016 * current^2 =
voltage
```

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). 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) $-\log_2 \frac{1}{44} = 5.56$ bits

2) $-\log_2 \frac{1}{2000} * \frac{1}{4.5} = 13.14$ bits

3) Unsure how to answer since incomplete question? Maybe we get 20 more points per character and then

$-\log_2 \frac{1}{2000} * \frac{1}{4.5} * \frac{1}{29} = 17.46$ bits

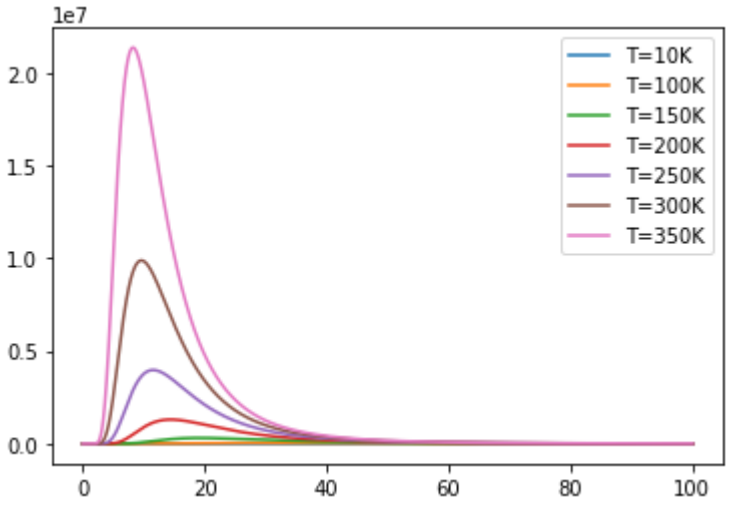
5. Wien's Approximation for the Temperature (bonus)

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.

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

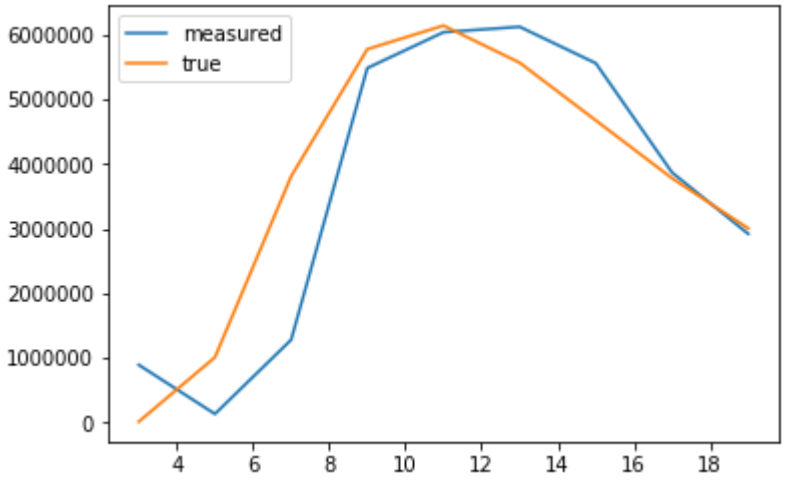
That is, the amount of energy depends on the fifth power of the wavelength λ and the temperature T of the body. The latter ensures a cutoff beyond a temperature-characteristic peak. Let us define this and plot it.

In [12]:
```
# 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 [13]:
```
# 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.