7/10

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 [47]:
```
from mxnet import nd, autograd, gluon
import matplotlib.pyplot as plt
import random
```

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 $o \to -\infty$.

Answers 1.1-1.4

1.1 By the properties of exponentials, we know that $\exp(o + c) = \exp(o)\exp(c)$, and similarly $\sum_{y'} \exp(o_{y'} + c) = \exp(c) \sum_{y'} (\exp(o_{y'}))$, hence adding a constant c gives the same probability estimate.

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

✓

,

1.2 For $y = 1$,

this is just a number
should be $\frac{1}{1+\exp(-o)}$, function of o.

$$\frac{\exp(o_y)}{\sum_{y'} \exp(o_{y'})} = \frac{e^1}{e^1 + e^0} = \frac{1}{1 + e^{-1}}$$

-1

1.3 Take log on $p(y = -1|o)^{-1}$ we have:

$$\log(\frac{e^1 + e^0}{e^0}) = \log(1 + \exp(1)) = \log(1 + \exp(-yo))$$

Take log on $p(y = 1|o)^{-1}$ we have:

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

1.4 Take limit on $-\log(p(1|o)) = \log(1 + \exp(-o))$ calculated above:

$$\lim_{o \to \infty} \log(1 + \exp(-o)) = \log(1) = 0$$

$$\lim_{o \to -\infty} \log(1 + \exp(-o)) = \log(1 + \infty) = \infty \to (-o)$$

2. Logistic Regression and Autograd

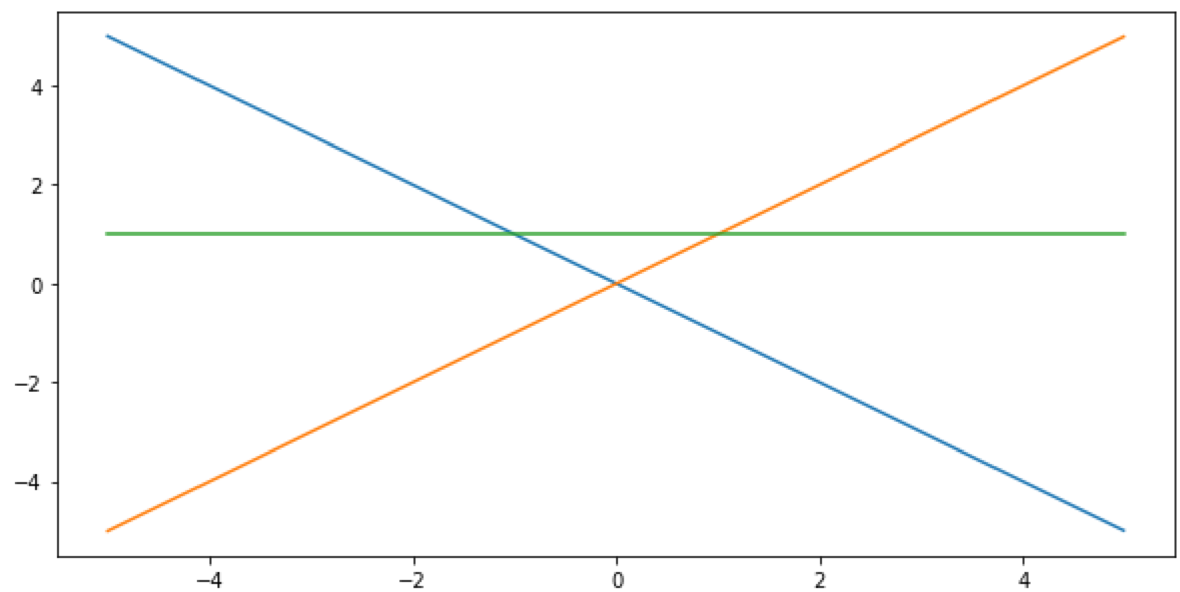
1. Implement the binary logistic loss $l(y, o) = \log(1 + \exp(-y \cdot o))$ in Gluon
2. Plot its values for $y \in \{-1, 1\}$ over the range of $o \in [-5, 5]$.
3. Plot its derivative with respect to o for $o \in [-5, 5]$ using 'autograd'.

In [2]:

```
## Q2.1
def loss(y,o):
    ## add your loss function here
    y = -y*o
    result = nd.exp(y)
    result = nd.log(result)
    return result
## Q2.2, 2.3
o = nd.arange(-5, 5, 0.01)
o.attach_grad()
with autograd.record():
    l_1 = loss(1, o)
    l_2 = loss(-1, o)
l_1.backward()
l_2.backward()


plt.figure(figsize=(10, 5))
plt.plot(o.asnumpy(), l_1.asnumpy())
plt.plot(o.asnumpy(), l_2.asnumpy())
plt.plot(o.asnumpy(), o.grad.asnumpy())
plt.show()
```

−2

← log(exp(-yo)) = -yo
we want log(1+exp(-yo)),
which has a non-constant
derivative in o.



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 [58]:
```
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])
## preps
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

def data_iter(batch_size, features, labels):
    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 ba
        # on the indices

def quad(X, a, b, c):
    return a*X**2+b*X+c

def linear(X, r, c):
    return r*X + c

def Ohm(X, r):
    return X*r

lr = 0.001  # Learning rate
num_epochs = 150  # Number of iterations
loss = squared_loss  # 0.5 (y-y')^2
batch_size = 3
data_size = len(voltage)

##3.1
net = quad
num_input = 3
a = nd.zeros(shape=(1,))
b = nd.zeros(shape=(1,))
c = nd.zeros(shape=(1,))

a.attach_grad()
b.attach_grad()
c.attach_grad()
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, current, voltage):
        with autograd.record():
            l = loss(net(X, a,b,c), y)
```

```
        l.backward()
        sgd([a,b,c], lr, batch_size)
    train = loss(net(X, a,b,c), y)
    result = train.mean().asnumpy()
    if result < 3:
        print('epoch', epoch + 1)
        print('loss is', result)
        break
    if epoch > 140:
        print('epoch %d, loss %f' % (epoch + 1, train.mean().asnumpy()))
print([a,b,c])
```

```
epoch 109
loss is [1.6929232]
[
[1.3191044]
<NDArray 1 @cpu(0)>,
[32.669792]
<NDArray 1 @cpu(0)>,
[12.381446]
<NDArray 1 @cpu(0)>]
```

In [73]:

```
## 3.2
net = linear
num_input = 2
r = nd.zeros(shape=(1,))
c = nd.zeros(shape=(1,))
lr = 0.01
batch_size = 7

r.attach_grad()
c.attach_grad()
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, current, voltage):
        with autograd.record():
            l = loss(net(X,r,c), y)
        l.backward()
        sgd([r,c], lr, batch_size)
    train = loss(net(X,r,c), y)
    result = train.mean().asnumpy()
    if result < 1:
        print('epoch', epoch + 1)
        print('loss is', result)
        break
    if epoch > 140:
        print('epoch %d, loss %f' % (epoch + 1, train.mean().asnumpy()))
print([r,c])
```

```
epoch 47
loss is [0.6849196]
[
[41.22986]
<NDArray 1 @cpu(0)>,
[5.251616]
<NDArray 1 @cpu(0)>]
```

In [77]:
```
## 3.3
net = Ohm
num_input = 1
r = nd.zeros(shape=(1,))
lr = 0.01
batch_size = 4

r.attach_grad()
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, current, voltage):
        with autograd.record():
            l = loss(net(X,r), y)
        l.backward()
        sgd([r], lr, batch_size)
    train = loss(net(X,r), y)
    result = train.mean().asnumpy()
    if result < 0.01:
        print('epoch', epoch + 1)
        print('loss is', result)
        break
    if epoch > 140:
        print('epoch %d, loss %f' % (epoch + 1, train.mean().asnumpy()))
print(r)
```

```
epoch 142, loss 0.556908
epoch 143, loss 0.655937
epoch 144, loss 0.320778
epoch 145, loss 0.985399
epoch 146, loss 0.063157
epoch 147, loss 0.684407
epoch 148, loss 0.610117
epoch 149, loss 1.109592
epoch 150, loss 0.282102

[42.03751]
<NDArray 1 @cpu(0)>
```

Q3 Obeservation:

with the same number of iterations, Ohm's Law obviously gives the most accurate result.

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.

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

should have been
$PPL(x) = \left(\frac{1}{p(x)}\right)^{1/len(x)}$

Answers for Q4

4.1 Since the monkey is typing completely randomly, each bit x is equally likely to be any of the 44 characters, and each bit is independent from all others. Hence

$$H(x) = \Sigma_{i=1}44(-p_i)\log_2(p_i) = 4\frac{1}{44}\log_2(44) = \log_2(44) \approx 2.45$$

✓

4.2 Each word, or on average every 4.5 letters, is equally likely to be any one of the 2000 words. Therefore $H(w) = \log_2(2000)$, each bit has $\frac{\log_2(2000)}{4.5} \approx 2.437$.

✓

4.3 (The question looks incomplete so I'm not sure what to answer..)

see solution.

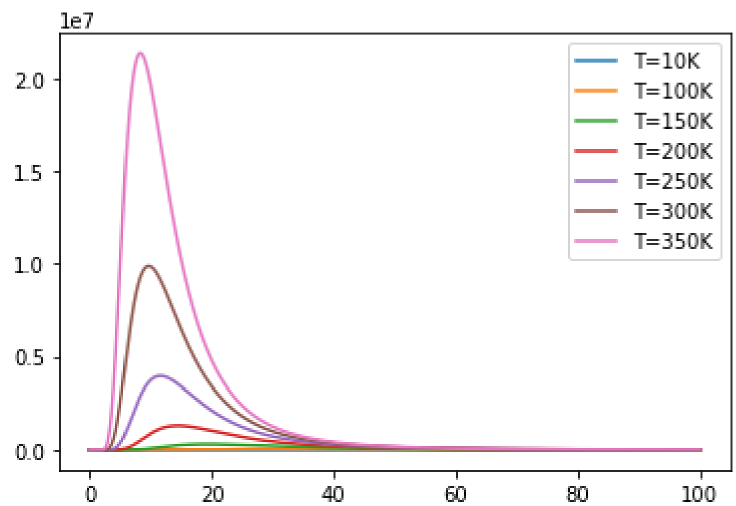
5. Wien's Approximation for the Temperature (bonus)

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.

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

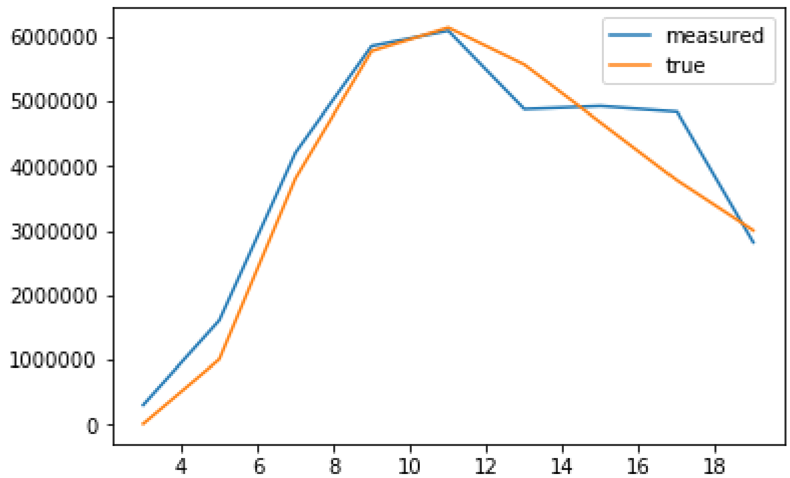
That is, the amount of energy depends on the fifth power of the wavelength λ and the temperature T of the body. The latter ensures a cutoff beyond a temperature-characteristic peak. Let us define this and plot it.

In [4]:
```
# 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 [5]:
```
# 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(), labe
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.