Homework 2 - Berkeley STAT 157

11/18

Handout 1/29/2019, due 2/5/2019 by 4pm in Git by committing to your repository.

In [295]:
```
from mxnet import nd, autograd, gluon
```

1. Multinomial Sampling

Implement a sampler from a discrete distribution from scratch, mimicking the function `mxnet.ndarray.random.multinomial` . Its arguments should be a vector of probabilities p. You can assume that the probabilities are normalized, i.e. tha they sum up to 1. Make the call signature as follows:

```
samples = sampler(probs, shape)

probs   : An ndarray vector of size n of nonnegative numbers summing up to 1
shape   : A list of dimensions for the output
samples : Samples from probs with shape matching shape
```

Hints:

1. Use `mxnet.ndarray.random.uniform` to get a sample from $U[0, 1]$.
2. You can simplify things for `probs` by computing the cumulative sum over `probs` .

In [296]:
```
def sampler(probs, shape):

    shape_holder = 1
    for i in shape:
        shape_holder = shape_holder * i

    probs_cum = np.cumsum(probs)


    starter = False
    for i in np.arange(shape_holder):

        #print(probs)
        pick = nd.random.uniform()
        #print(pick)

        for i in np.arange(1, len(probs_cum)+1):
            if pick < probs_cum[0]:
                if starter is False:
                    em = nd.array([0])
                    starter = True
                else:
                    em = nd.concat(em, nd.array([0]), dim = 0)
                #print(0)
                break
            else:
                if probs_cum[i-1] <= pick < probs_cum[i]:
                    if starter is False:
                        em = nd.array([i])
                        starter = True
                    else:
                        em = nd.concat(em, nd.array([i]), dim = 0)
                    #print(i)
                    break
    return em.reshape(shape)


# a simple test
sampler(nd.array([0.2, 0.3, 0.5]), (2,3))
```

3/4
see solution

Out[296]:
```
[[1. 0. 2.]
 [1. 2. 1.]]
<NDArray 2x3 @cpu(0)>
```

2. Central Limit Theorem

Let's explore the Central Limit Theorem when applied to text processing.

- Download https://www.gutenberg.org/ebooks/84 (https://www.gutenberg.org/files/84/84-0.txt) from Project Gutenberg
- Remove punctuation, uppercase / lowercase, and split the text up into individual tokens (words).
- For the words `a` , `and` , `the` , `i` , `is` compute their respective counts as the book progresses, i.e.

$$n_{\text{the}}[i] = \sum_{j=1}^{i} \{w_j = \text{the}\}$$

- Plot the proportions $n_{\text{word}}[i]/i$ over the document in one plot.
- Find an envelope of the shape $O(1/\sqrt{i})$ for each of these five words.
- Why can we **not** apply the Central Limit Theorem directly?
- How would we have to change the text for it to apply?
- Why does it still work quite well?

0/4 plots?

In [297]:
```python
filename = gluon.utils.download('https://www.gutenberg.org/files/84/84-0.txt')
with open(filename) as f:
    book = f.read()
print(book[0:100])


import re
book = book.lower()
book = re.sub(r'[^\w\s]','',book)
book = book.split()

a = book.count('a')/len(book),
i = book.count('i')/len(book),
an = book.count('and')/len(book),
the = book.count('the')/len(book)

## Add your codes here
```

```
Project Gutenberg's Frankenstein, by Mary Wollstonecraft (Godwin) Shell
ey

This eBook is for the u
```

The central limit theorem needs all of the proportions of every word to get the full picture. We only selected a couple words, albeit common, but doesn't really make CLT applicable. In a way the test is sequential, but since there's still a ridiculous amount of samples so the CLT is still able to be seen.

4/4

We would need to take out a lot of the words in the text that aren't the ones that we chose.

3. Denominator-layout notation

We used the numerator-layout notation for matrix calculus in class, now let's examine the denominator-layout notation.

Given $x, y \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$, we have

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}, \quad \frac{\partial \mathbf{y}}{\partial x} = \left[\frac{\partial y_1}{\partial x}, \frac{\partial y_2}{\partial x}, \ldots, \frac{\partial y_m}{\partial x} \right]$$

and

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{y}}{\partial x_1} \\ \frac{\partial \mathbf{y}}{\partial x_2} \\ \vdots \\ \frac{\partial \mathbf{y}}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1}, \frac{\partial y_2}{\partial x_1}, \ldots, \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2}, \frac{\partial y_2}{\partial x_2}, \ldots, \frac{\partial y_m}{\partial x_2} \\ \vdots \\ \frac{\partial y_1}{\partial x_n}, \frac{\partial y_2}{\partial x_n}, \ldots, \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Questions:

1. Assume $\mathbf{y} = f(\mathbf{u})$ and $\mathbf{u} = g(\mathbf{x})$, write down the chain rule for $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$
2. Given $\mathbf{X} \in \mathbb{R}^{m \times n}$, $\mathbf{w} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, assume $z = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$, compute $\frac{\partial z}{\partial \mathbf{w}}$.

2/3
see solution

1. dy/dx = (dy/du)(du/dx)

1. dz/dw = (dz/db)(db/da)(da/dw) = 2 *(b transpose)* I *X = 2* ((Xw - y) transpose) * (X transpose)

4. Numerical Precision

Given scalars `x` and `y`, implement the following `log_exp` function such that it returns a numerically stable version of

$$-\log\left(\frac{e^x}{e^x + e^y}\right)$$

In [298]:
```
def log_exp(x, y):

    return -nd.log(nd.exp(x)/(nd.exp(x) + nd.exp(y)))
```

Test your codes with normal inputs:

In [299]:
```
x, y = nd.array([2]), nd.array([3])
z = log_exp(x, y)
z
```

Out[299]:
```
[1.3132617]
<NDArray 1 @cpu(0)>
```

Now implement a function to compute $\partial z / \partial x$ and $\partial z / \partial y$ with `autograd`

In [300]:
```
def grad(forward_func, x, y):
    x.attach_grad()
    y.attach_grad()
    with autograd.record():
        z = forward_func(x, y)
        print(z)
    z.backward()
```

In [ ]:
```
```

Test your codes, it should print the results nicely.

In [301]:
```
grad(log_exp, x, y)
```

```
[1.3132617]
<NDArray 1 @cpu(0)>
```

But now let's try some "hard" inputs

In [302]:
```
x, y = nd.array([50]), nd.array([100])
grad(log_exp, x, y)
```

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

Does your code return correct results? If not, try to understand the reason. (Hint, evaluate `exp(100)`). Now develop a new function `stable_log_exp` that is identical to `log_exp` in math, but returns a more numerical stable result.

In [9]:
```
def stable_log_exp(x, y):
    ## Add your codes here
    pass

grad(stable_log_exp, x, y)
```

2/3

```
x.grad = None
y.grad = None
```

In [ ]: