Homework 2 - Berkeley STAT 157

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

In [3]:
```
from mxnet import nd, autograd, gluon
import numpy as np
import mxnet as mx
import string
from matplotlib import pyplot as plt
import random
```

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. that 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 [3]:
```
def sampler(probs, shape):
    ## Add your codes here
    if isinstance(shape, int):
        length = shape
        width = 1
        switch = True
    else:
        length = shape[1]
        width = shape[0]
        switch = False
    toReturn = nd.zeros(shape)
    cumsum = np.cumsum(probs)
    def roll(samp):
        j = 0
        for i in range(length):
            if samp <= cumsum[i]:
                return j
            else:
                j += 1
    for i in range(width):
        rand = mx.ndarray.random.uniform(shape=(length,))
        arr = []
        for ra in rand:
            arr.append(roll(ra))
        if switch:
            toReturn = nd.array(arr)
        else:
            toReturn[i,] = nd.array(arr)
    return toReturn

# a simple test
sampler(nd.array([0.2, 0.3, 0.5]), (2,3))
# more test
sampler(nd.array([0.2, 0.2, 0.6]), 6)
# more test
sampler(nd.array([0.1, 0.2,0.2,0.5]), (3,4))
```

Out[3]:
```
[[2. 1. 3. 0.]
 [3. 1. 2. 2.]
 [3. 3. 3. 2.]]
<NDArray 3x4 @cpu(0)>
```

2. Central Limit Theorem

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

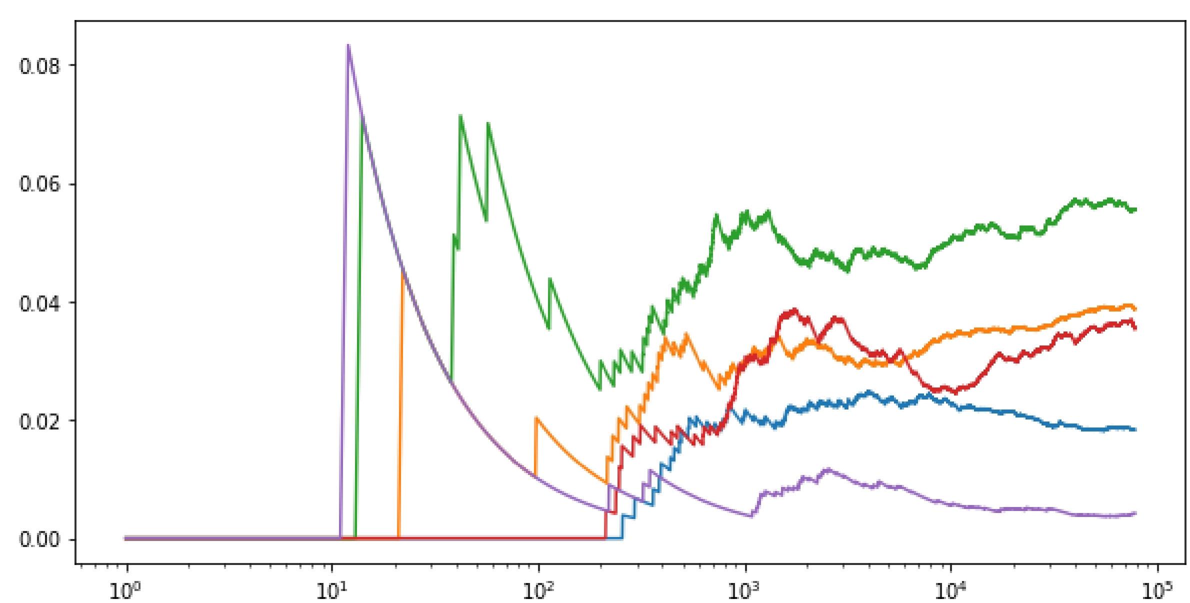
- Download https://www.gutenberg.org/ebooks/84 (https://www.gutenberg.org/files/84/84-0.txt) from Project Gutenberg
- Remove punctuation, uppercase / lowercase, and split the text up into individual tokens (words).
- For the words `a`, `and`, `the`, `i`, `is` compute their respective counts as the book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

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

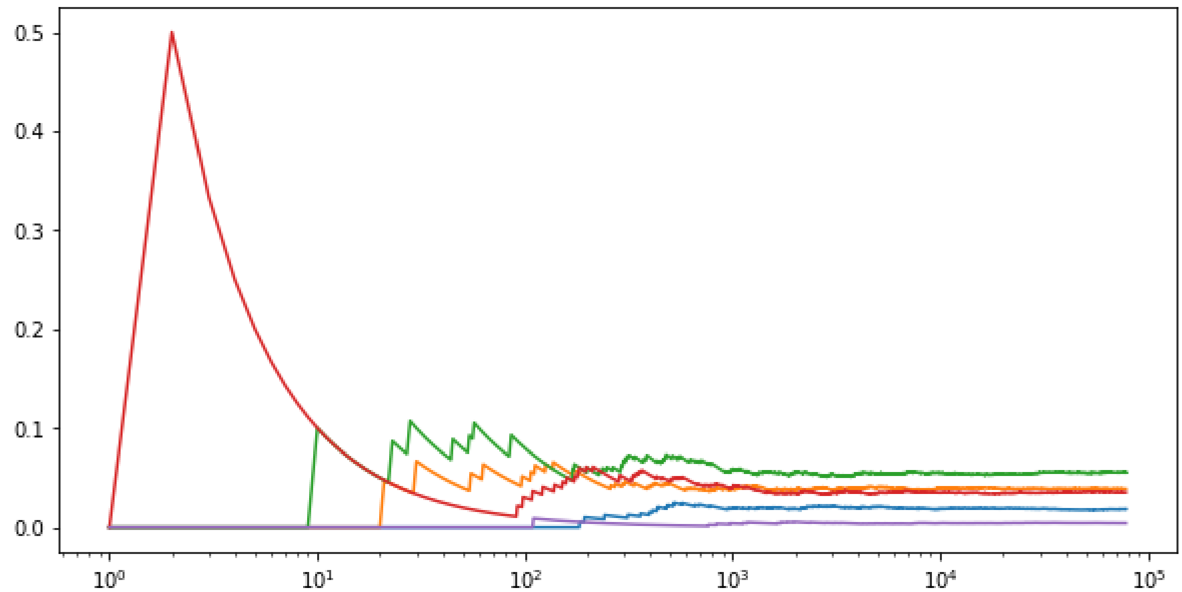
- Plot the proportions $n_{\text{word}}[i]/i$ over the document in one plot.
- Find an envelope of the shape $O(1/\sqrt{i})$ for each of these five words.
- Why can we **not** apply the Central Limit Theorem directly?
- How would we have to change the text for it to apply?
- Why does it still work quite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

In [45]:

```
#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])

book = book.lower()
translator = str.maketrans('', '', string.punctuation)
book = book.translate(translator)
book = book.replace("\n", " ")
book = book.split()
total = len(book)
## Add your codes here
y = np.arange(1,total+1)

def count(wordlist,word):
    x = np.zeros(total)
    for i in range(total):
        if wordlist[i] == word:
            x[i] += 1
            x[i:] = x[i]
    return x/y
a = count(book,'a')
b = count(book,'and')
c = count(book,'the')
d = count(book,'i')
e = count(book,'is')

plt.figure(figsize=(10,5))
for i in [a,b,c,d,e]:
    plt.semilogx(y,i)
plt.show()

#Why can we not apply the Central Limit Theorem directly?
# The occurance of the words are not independent and identically distributed for CLT to apply
```



In [46]:
```
#How would we have to change the text for it to apply?
# Randomdize the word list
random.shuffle(book)
a = count(book,'a')
b = count(book,'and')
c = count(book,'the')
d = count(book,'i')
e = count(book,'is')

plt.figure(figsize=(10,5))
for i in [a,b,c,d,e]:
    plt.semilogx(y,i)
plt.show()

#Why does it still work quite well?
# Disorder the original list, so that the occurance of each word is appr
oximately independent
```



In [43]:
```
countArray = np.zeros(5)
countArray[0] = book.count("a")
countArray[1] = book.count("and")
countArray[2] = book.count("the")
countArray[3] = book.count("i")
countArray[4] = book.count("is")
print(countArray)
total = len(book)
countArray = countArray/total
char = ["a","and","the","i","is"]
print(char)
print(countArray)
# plt.figure(figsize=(10,5))
# plt.bar(char,countArray)
# plt.show()
```

```
[1439. 3028. 4329. 2766.  330.]
['a', 'and', 'the', 'i', 'is']
[0.01842911 0.03877925 0.05544101 0.03542384 0.00422627]
```

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

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

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 $\frac{\partial z}{\partial \mathbf{w}}$.

1:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{u}} \cdot \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

2:

$$\text{decompose:}$$
$$\mathbf{a} = \mathbf{X}\mathbf{w}$$
$$\mathbf{b} = \mathbf{a} - \mathbf{y}$$
$$z = ||\mathbf{b}||^2$$

$$\frac{\partial z}{\partial \mathbf{w}} = \frac{\partial z}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{w}} = 2\mathbf{b}^T \times \mathbf{I} \times \mathbf{X}$$
$$= 2(\mathbf{X}\mathbf{w} - \mathbf{y})^T \mathbf{X} = 2\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

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 [4]:
```
def log_exp(x, y):
    ## add your solution here
    return -nd.log(np.e**x/(np.e**x+np.e**y))
```

Test your codes with normal inputs:

In [5]:
```
x, y = nd.array([2]), nd.array([3])
z = log_exp(x, y)
z
```

Out[5]:
```
[1.3132616]
<NDArray 1 @cpu(0)>
```

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

In [6]:
```
def grad(forward_func, x, y):
    ## Add your codes here
    x.attach_grad()
    y.attach_grad()
    with autograd.record():
        z = forward_func(x,y)
    z.backward()
    print('x.grad =', x.grad)
    print('y.grad =', y.grad)
```

Test your codes, it should print the results nicely.

In [7]:
```
grad(log_exp, x, y)
```

```
x.grad =
[-0.7310586]
<NDArray 1 @cpu(0)>
y.grad =
[0.7310586]
<NDArray 1 @cpu(0)>
```

But now let's try some "hard" inputs

In [8]:
```
x, y = nd.array([50]), nd.array([100])
grad(log_exp, x, y)
```

```
x.grad =
[nan]
<NDArray 1 @cpu(0)>
y.grad =
[nan]
<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):
    return nd.log(1+np.e**(y-x))
grad(stable_log_exp, x, y)
```

```
x.grad =
[-0.9999999]
<NDArray 1 @cpu(0)>
y.grad =
[0.9999999]
<NDArray 1 @cpu(0)>
```