Simulating Language: Lab 8 Worksheet

Download `bayes1.py` from the usual place. This simulation features a replication of the Reali & Griffiths iterated learning model of the evolution of frequency distributions, and is built around a Bayesian model of inference. This simulation allows you to explore the effects of learning bias on learning and cultural evolution, and also gives you your first chance to see under the hood of a Bayesian model. But before we get onto the model itself, we need to talk about log probabilities.

Introduction to log probabilities

In the lectures I introduced Bayes' Rule as a relationship between probabilities: the posterior is proportional to the product of the likelihood and the prior, and all three of these quantities are probabilities. Doing Bayesian models of learning therefore involves manipulating probabilities, numbers between 0 and 1. And some of these probabilities can be very small indeed, because they involve multiplying small numbers lots of times (consider, for instance, how small the probability is of getting 100 heads if you flip a fair coin 100 times: it's 0.5 x 0.5 x 0.5 ... 100 times, or 0.5^{100} if you prefer. That's a very small number.

Working with small numbers on a computer can be a problem, because the computer cannot exactly represent real numbers (i.e. numbers we would write in decimal notation, e.g. numbers like 0.1, 3.147). Your computer has a very large memory where it can store and manipulate numbers, but the problem is that this memory is necessarily finite (it has to fit in your computer) and there are infinitely many real numbers. Think of recurring decimal you get by dividing 1 by 3, 0.3333..., where the threes go on forever - it would take an infinite amount of space to exactly represent this number in your computer, and distinguish it from a very similar number, e.g. 0.33333... where the threes go on for a few thousand repetitions only. So there's no way your computer can exactly represent every possible real number. What it does instead is store numbers as accurately as it can, which involves introducing small rounding errors - you can see in the box on the right that these rounding errors crop up even in places where you might think it would be easy for the computer to store the correct number, but these 'errors' are just a result of the compromise your computer has to make when it works with real numbers. In fact your computer does its best to conceal these errors from you, and often displays numbers in a format that hides exactly what numbers it is actually working with.

```
In [1]: 0.1
Out[1]: 0.1
In [2]: 0.2
Out[2]: 0.2
In [3]: 0.1 + 0.2
Out[3]: 0.30000000000000004
```

Why do you need to care about this? Well, if you are dealing with very very small numbers (as you might do if you were doing a Bayesian model which involves learning from lots of data) then the rounding errors become a real factor - for big numbers the rounding errors are so small we don't really care, but for very small numbers, the rounding errors might be relatively big. Worse, sometimes the computer will round a very very small number to 0, which can generate unexpected and hard-to-predict errors in your code (e.g. if you try to divide something by a very very small number which gets rounded to 0).

The solution to this is to have the computer work not with probabilities, but with *log probabilities*: we take our probabilities, take the log of those numbers, then carry on as before. As you can see

```
In [4]: log(1)
Out[4]: 0.0
In [5]: log(0.1)
Out[5]: -2.3025850929940455
In [6]: log(0.000001)
Out[6]: -13.815510557964274
In [7]: exp(log(0.5))
Out[7]: 0.5
In [8]: exp(log(0.1))
Out[8]: 0.10000000000000002
```

from the box on the left, taking the log of a very small number turns it into a large negative number - these are still real numbers, so the computer still can't represent them exactly, but in the log domain the rounding errors will be proportionately smaller for very small numbers and the rounding-to-0 problem won't crop up. Then, if we want to see an actual probability, rather than a log probability, we can reverse this process, using the `exp` function, to get back raw probabilities. Jumping back and forth from logs can introduce rounding errors of its own (see code line 8 in the box), but it's necessary to avoid the catastrophic rounding errors you can get if you just work with raw probabilities.

```
In [9]: 0.5 * 0.5
Out[9]: 0.25
In [10]: exp(log(0.5) + log(0.5))
Out[10]: 0.25
In [11]: 0.5 / 0.5
Out[11]: 1.0
In [12]: exp(log(0.5) - log(0.5))
Out[12]: 1.0
```

Some basic arithmetic operations work a bit differently with logs. If you want to multiply two probabilities, you add their logarithms; if you want to divide one probability by another, you subtract the logarithm of one from another. And there is no direct equivalent of adding and subtracting in the log domain, which involves a little bit of fancy footwork in the code that you don't have to worry about too much. The important thing is 1) to understand that the code is going to manipulate log probabilities and 2) this changes nothign conceptually, it's just a matter of implementation.

On to the code

The code starts with various bits and pieces which we need for working with logs and probability distributions. In particular, it loads in a function called `logsumexp` which allows us to do addition in the log domain (remember, just using the normal addition operator + with logs is the equivalent of multiplying the non-logs). Then there is a function called `logsubtract` which allows us to do the equivalent of subtraction in the log domain (because if we just use normal subtraction, -, that's equivalent to division). Then there are a couple of functions which we need for doing probabilistic sampling the log domain - `normalize_logprobs` will take a list of logs and normalise them for us (the equivalent of taking a list of pseudo-probabilities and rescaling them so they sum to 1, but in the log domain) and `log_roulette_wheel` takes a list of log probabilities and selects a random index from that list, with probability of any particular index being selected being given by its log probability. These functions are used elsewhere in the code, but it is not important that you understand exactly how they work.

The main part of the code starts by setting up the grid. As discussed in class, we are going to turn a problem of inferring a potentially continuous value (the probability with which your teacher uses word 1) into a problem of inferring one of a limited set of possible values (either your teacher is using the word with probability 0.005, or 0.015, or 0.025, etc). In the code we will refer to a certain probability of using word 1 as pW1. We will call this set of possible values for pW1 *the grid* - you can set the granularity of the grid as high as you like, but 100 works OK without being too slow. We

are actually going to maintain two grids - one of probabilities, and one of log probabilities (since we are going to work with log probabilities when we do our calculations).

```
# ----- setting up the grid -----
grid_granularity = 100
grid_increment = 1/(grid_granularity+0.)

#sets up the grid of possible probabilities to consider
possible_pW1 = []
for i in range(grid_granularity):
    possible_pW1.append(grid_increment/2 + (grid_increment*i))

#sets up the grid of log probabilities
possible_logpW1 = []
for pW1 in possible_pW1:
    possible_logpW1.append(log(pW1))
```

Have a look at the two grids (`possible_pW1` and `possible_logpW1`). Do they look like you expected?

Next up come the various functions we need for Bayesian inference. I will step through these gradually.

The prior

```
# ----- functions for Bayesian inference -----

def calculate_prior(alpha):
    '''
    Calculates the prior probability of all values of possible_pw1.
    Only produces symmetrical priors: favours regularity when alpha < 1,
    uniform when alpha = 1, favours variability when alpha > 1.
    NOTE that this function is not called anywhere else in this code - you
    can use it to display a prior if you like.
    '''
    logprior = []
    for pW1 in possible_pW1:
        logprior.append(beta.pdf(pW1,alpha,alpha))
    return normalize_probs(logprior)

def calculate_logprior(alpha):
    '''
    Calculates the log prior probability of all values of possible_pw1.
    '''
    logprior = []
    for pW1 in possible_pW1:
        logprior.append(beta.logpdf(pW1,alpha,alpha))
    return normalize_logprobs(logprior)
```

There are two functions for calculating the prior probability distribution, the prior probability of each of our possible values of pW1. One of these returns raw probabilities, so you can look at the prior easily without worrying about logs. The second, which is the one our code actually uses, calculates the log probability distribution - i.e. it deals with log probabilities, not logs. The beta

distribution, which is what we are using for our prior, is a standard probability distribution, so we can just use a function from a library (`beta.pdf` for raw probabilities, `beta.logpdf` for log probabilities) to get the probability density for each value of pW1, then normalise those to convert them to probabilities.

Plot some different prior probability distributions - for example, try typing
`plot(possible_pW1,calculate_prior(0.1))`
at the prompt, to see the prior probability distribution over various values of pW1 for the alpha=0.1 prior. What values of alpha lead to a prior bias for regularity? What values of alpha lead to a prior bias for variability? What values of alpha lead to a completely unbiased learner?

Likelihood and production

In order to do Bayesian inference, we need a likelihood function that tells us how probable a set of data is given a certain hypothesis (a value of pW1). And to do iterated learning we need a way of modelling production - taking an individual, with a value of pW1 in their head, and having them produce data that someone else can learn from. The next two functions do that job.

```
def likelihood(data,logpW1):
    '''Calculates the log probability of data d, where data is a string of 0s
    (representing word 0) and 1s (representing word 1)'''
    logpW0 = log_subtract(log(1),logpW1) #probability of w0 is 1-prob of w1
    logprobs = [logpW0,logpW1]
    loglikelihoods = []
    for d in data:
        loglikelihood_this_item = logprobs[d] #d will be either 0 or 1,
                                              #so can use as index
        loglikelihoods.append(loglikelihood_this_item)
    return sum(loglikelihoods) #summing log probabilities =
                               #multiply non-log probabilities

def produce(logpW1,n_productions):
    '''
    Returns data, a list of 0s and 1s (representing w0 and w1)
    '''
    logpW0 = log_subtract(log(1),logpW1)
    logprobs = [logpW0,logpW1]
    data = []
    for p in range(n_productions):
        data.append(log_roulette_wheel(logprobs))
    return data
```

We are going to model data - sets of utterances - as a simple list of 0s and 1s: the 0s correspond to occurrences of word 0, the 1s correspond to occurrences of w1. Both functions take a (log) probability of w1 being produced, and use that to calculate the probability of w0 (which is 1 minus the probability of word 1).

Test out the produce function - remember, you need to feed it a log probability, so decide on a probability for w1 and then convert it to log using the `log` *function. What kind of data will be produced if the probability of w1 is low? Or if it is high? Next, check out the likelihood function - how does the likelihood of a set of data depend on the data and the probability of word 1? Remember that the likelihood function returns a log probability, so you can convert this to a probability using the* `exp` *function.*

Learning

Now we have all the bits we need to calculate the posterior probability distribution, and therefore to do learning (by picking a hypothesis, a value of pW1, based on its posterior probability).

```
def posterior(data,prior):
    '''
    Calculates posterior probability for all possible values of logpW1, given
    data and prior (a list of log probabilities). Considers the values of
    logpW1 given in the list possible_logpW1.
    '''
    posterior_logprobs = []
    for i in range(len(possible_logpW1)):
        logpW1 = possible_logpW1[i]
        logp_h = prior[i] #prior probability of this pW1
        logp_d = likelihood(data,logpW1) #likelihood of data given this pW1
        posterior_logprobs.append(logp_h + logp_d) #adding logs =
                                                   #multiplying non-logs
    return normalize_logprobs(posterior_logprobs)


def learn(data,prior):
    '''
    Infers the (log) probability of word 1, given prior and data:
    calculates posterior probability distribution, then selects a value using
    log_roulette_wheel.
    '''
    posterior_logprobs = posterior(data,prior)
    selected_index = log_roulette_wheel(posterior_logprobs)
    return possible_logpW1[selected_index]
```

Test out the learn function. To do this you will need to build a prior, and some data - the box to the right shows you how, for a uniform prior (alpha = 1) and data consisting of two 1s and two 0s (note that there is a cute little trick there for creating lists of duplicates and sticking two lists together). Start with a uniform prior and see how the data affects the learner's hypothesis. What does adding more data do? What does making the data highly skewed in favour of one word do? Then try different priors - what does a strong prior in favour of regularity do? What does a strong prior in favour of variability do?

```
In [13]: my_prior =
calculate_logprior(1)
In [14]: [0] * 2
Out[14]: [0, 0]
In [15]: [0] + [1]
Out[15]: [0, 1]
In [16]: my_data = [0]*2 + [1]*2
In [17]: my_data
Out[17]: [0, 0, 1, 1]
In [18]: exp(learn(my_data,my_prior))
Out[18]: 0.5750000000000001
In [19]: exp(learn(my_data,my_prior))
Out[19]: 0.685
```

Iteration

At last, we have all the bits we need to do iterated learning: we can have a learner infer a value of pW1 given some observed data, then we can have that individual produce data which another

individual can learn from. **Look at the usage example at the top of bayes1.py to see how to run an iterated learning simulation using this code!**

```
# ----- iterated learning -----

def iterate(alpha,n_productions,starting_count_w1,generations):
    '''
    Runs an iterated learning simulation.
    Starts with data consisting of starting_count_w1 instances of w1 and
    (n_productions-starting_count_w1) instances of w0.
    Returns two values: the inferred probability of w1 at each generation
    from 1 onwards (not a log - I convert it to a genuine probability for you),
    and the number of productions of w1 from generation 0 onwards.
    '''
    prior = calculate_logprior(alpha)
    pW1_accumulator=[nan]
    data_accumulator=[starting_count_w1]
    data=[1]*starting_count_w1 + [0]*(n_productions-starting_count_w1)
    for generation in range(1,generations+1):
        logpW1 = learn(data,prior)
        data=produce(logpW1,n_productions)
        pW1_accumulator.append(exp(logpW1))
        data_accumulator.append(sum(data))
    return pW1_accumulator,data_accumulator
```

Questions

The priority for this worksheet is to work through the in-text questions above: experimenting with the prior, checking that the likelihood and production makes sense, checking you understand how learning depends on the prior and the data. Once you are happy with that, try these questions:

1. One of Reali & Griffiths's main points was that studying learning in a single individual can be a bad way to discover their prior bias, particularly if you give them lots of data which swamps this prior bias - given enough data, learners with quite different priors look the same. Can you reproduce this effect using this code?
2. Iterated learning can potentially give a clearer picture of prior bias. Try running some simulations for 10 generations, with 10 data points passed from generation to generation, starting each simulation with 5 instances of w1 and 5 of w0. How does changing the prior change the results? Try alpha=0.1, alpha=1, and alpha=5. Are the differences between different priors obvious after generation 1, or do they become more apparent over generations?
3. Now try messing with the amount of data that is passed from generation to generation. What happens if you pass more data between generations? What happens if you pass less? What happens if you pass *no data* from generation to generation? What would this latter setting correspond to in the real world?