Simulating Language: Lab 9 Worksheet

Download `bayes_U18.py` from the usual place.This simulation features a replication of the Culbertson & Smolensky model of learning biases for Greenberg's Universal 18.

	N-Adj	Adj-N
Num-N	17%	27%
N-Num	52%	4%

Figure 1: Universal 18: typology

The code...

Overview

The model uses Bayesian inference to predict the types of grammars learners will infer given (1) a set of counts of [Adj-N, N-Adj, Num-N, N-Num] utterances and (2) their prior expectations in terms of variation and ordering combinations. The first part of code imports what we'll need to use the binomial and beta distributions, to do logs and exponentials, and to generate random numbers of various kinds.

The input data

The input consists of counts of [Adj-N, N-Adj, Num-N, N-Num] with 40 total per modifier type. Whether the input is skewed toward pre- or post-nominal modifiers for each phrase type depends on the condition, as shown in the table.

As in your previous lab, we'll use a grid of probabilities to describe the space of possible generating grammars. This time, we'll need to use the grid twice–once for the probability of Adj-N (vs N-Adj), and again for the probability of Num-N (vs. N-Num).

Training counts:
condition 1: [28,12,28,12]
condition 2: [12,28,12,28]
condition 3: [12,28,28,12]
condition 4: [28,12,12,28]

→ *Check out `training_data`. How would you access just the counts for condition 2? Also take a look at the grid. How is it different from the one you used in the previous lab?*

```
#Training data is a list of lists of counts for each condition
#                pattern 1     pattern 2     pattern 3     pattern 4
#               (Adj-N,Num-N) (N-Adj,N-Num) (N-Adj,Num-N) (Adj-N,N-Num)
training_data = [[28,12,28,12],[12,28,12,28],[12,28,28,12],[28,12,12,28]]

# set up the grid
grid_granularity = 100 # granularity of grid
possible_p = []
for i in range(1,grid_granularity):
    possible_p.append(i/(grid_granularity+0.))
```

Likelihood

The likelihood here is calculated using the binomial distribution, the same you used to calculated the likelihood of a particular sequence of word 0's and word 1's in the previous lab. Only difference is that here we don't care what order they were heard in, just about the number of Adj-N out of all Adj trials, and the number of Num-N out of all Num trials.

The binomial distribution is the same you'd use to figure out the likelihood of some number of heads $counts_H$ out of t coin tosses, for a fair or biased coin.

$$binomial(counts_H|p_H,t) = \binom{t}{c} p^c(p-1)^{t-c}$$

→ *Check out `U18_likelihood`. Notice that it returns log probabilities. Calculate the likelihood of getting 28 Adj-N counts out of a total of 40 when the underlying probability of Adj-N according to the grammar is 0.7 compared to 0.3.*

```
def U18_likelihood(data,p_AdjN,p_NumN):
    '''Calculates log likelihood of data, (where data are counts representing
    number of Adj-N out of total Adj instances and
    number of Num-N out of total Num instances)
    given point probability of Adj-N, and Num-N
    '''
    loglikelihood = [];
    loglikelihood_AdjN = binom.logpmf(data[0],data[0]+data[1],p_AdjN)
    loglikelihood_NumN = binom.logpmf(data[2],data[2]+data[3],p_NumN)
    loglikelihood = loglikelihood_AdjN + loglikelihood_NumN
    return loglikelihood
```

Prior

THE PRIOR in this model has two parts. One part you know about already: regularization as encoded by the parameters of the beta distribution. In the last lab you used a single parameter, alpha, for a single symmetrical beta distribution. Here we want two separate asymmetrical beta distributions–favoring *either* probabilities close to one *or* close to zero, but not both.

The two parameters of the beta distribution are (annoyingly) called *alpha* and *beta*. To get the second part of the prior–the part which favors particular combinations of orders (i.e., Adj-N with Num-N) we need to promote or penalize particular parts of the two dimension grammar space. We can do this by defining four "components" using different combinations of *alpha* and *beta*, with *alpha* constrained to be higher than *beta*–Fig. 3 illustrates why. Then, when we calculate the prior probability of any grammar, it will actually be the sum of its probability given the beta distributions governing p(Adj-N) and p(Num-N) for each component, weighted by the probability of that component. The latter is determined by *g*. If the probability of a given component (e.g., `g[4]`) is low, then even grammars very likely to be generated by it will have a low prior probability.

```
def U18_prior(g, a, b, p_AdjN, p_NumN):
    '''
    Calculates the log prior probability of a given p_AdjN and p_NumN given a set
        of parameters.
    This is a sum over the probabilities given by each mixture component.
    Parameter are: g (set of four mixture weights), a(lpha), b(eta) (Beta shape
        parameters)
    '''
    pattern1_component = [a,b,a,b] # higher prob for Adj-N, Num-N
    pattern2_component = [b,a,b,a] # higher prob for N-Adj, N-Num
    pattern3_component = [b,a,a,b] # higher prob for N-Adj, Num-N
    pattern4_component = [a,b,b,a] # higher prob for Adj-N, N-Num
    components = [pattern1_component,pattern2_component,pattern3_component,
        pattern4_component]

    logprior=[]
    for i in range(0,4): # loop over all four components
        logprior_i_Adj = beta.logpdf(p_AdjN,components[i][0],components[i][1])
        logprior_i_Num = beta.logpdf(p_NumN,components[i][2],components[i][3])
        logprior_i = logprior_i_Adj + logprior_i_Num
        logprior.append(logprior_i)
    # a+b+... in log space = log(exp(a)+exp(b)+...)
    logprior = log((g[0]*exp(logprior[0])) + (g[1]*exp(logprior[1])) + (g[2]*exp(
        logprior[2])) + (g[3]*exp(logprior[3])))
    return logprior
```

The optimal prior parameters

THE PARAMETERS of the prior are *alpha, beta, g*. The first two (called *a*, and *b* in the code) encode the regularization bias. The higher *a* is relative to *b*, the more regularization. The third, *g* is a set of weights for the four components, summing to 1. These parameters were fit to the experimental data, yielding:

```
#                   gamma                   alpha, beta
fit_parameters = [[0.6293,0.3706,0.0001,0],16.5, 0.001]
```

→ *Why do we need asymmetrical beta distributions in this case?*

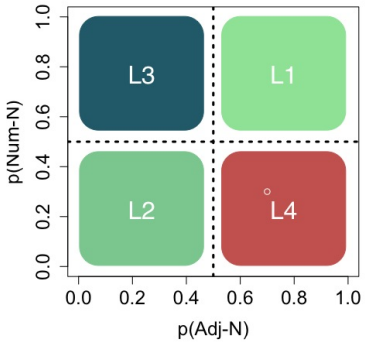

Figure 2: The four components

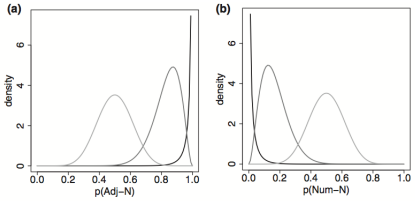

Figure 3: Example beta distributions with: (a=10, b=10), (a=15, b=3), (a=15, b=0.1) for Adj-N, and the reverse for Num-N

→ *Take a look at the four components in the prior. Can you see why they are defined by those combinations of alpha and beta?*

→ *Each component–a combination of two beta distributions–can in principle generate any grammar, that is any pair p(Adj-N), p(Num-N). Which component assigns the highest probability to the pair (0.8,0.8)? How about (0.4,0.9)?*

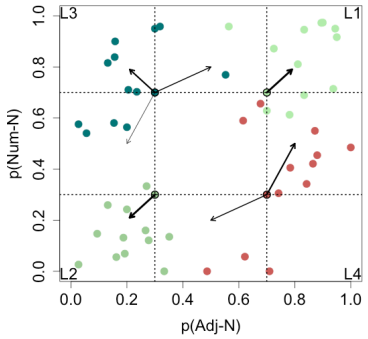

Figure 4: Individual learner results used to fit the model parameters.

Posterior

THE POSTERIOR distribution over grammars can now be calculated using the likelihood and the prior with the fit parameters.

```
def U18_posterior(g,a,b,data):
    '''Calculates the log posterior probability of a set of counts
    for all possible p_AdjN, p_NumN combinations,
    given prior parameters g, a(lpha), b(eta)
    '''
    posterior = []
    for p_a in range(len(possible_p)):
        for p_n in range(len(possible_p)):
            lik_i = U18_likelihood(data,possible_p[p_a],possible_p[p_n])
            prior_i = U18_prior(g,a,b,possible_p[p_a],possible_p[p_n])
            posterior.append(lik_i+prior_i)

    return posterior
```

→ *Given the fit parameters, will the model predict a weak or strong regularization bias? What does the model predict the posterior probability of a grammar like (0.8,0.2) will be? Why?*

Plot the results

THE MODEL can now be used to generate predicted learning outcomes; given some input data, and the prior parameters, we can generate sampled grammars and plot them in the two-dimensional grammar space we've be using. Fig. 5 shows this as reported in the original Culbertson & Smolensky paper. You can use the `U18_roulette_wheel` function to reproduce this. This function is similar to those you've previously used to sample probabilities. The difference is that this function samples *pairs* of p(Adj-N), p(Num-N) probabilities, and it lets you specify how many such samples you want. You'll notice there's also a normalization function `normalize_log_distribution` which takes log probabilities output by the posterior and makes them a proper (non-log) probability distribution for plotting.

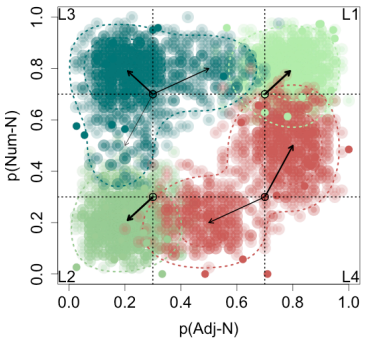

Figure 5: Distribution of grammars predicted by the model.

```
def U18_roulette_wheel(g,a,b,data,num_samps):
    '''Generates a random sample of grammars (p_AdjN, p_NumN pairs)
    with probability of selection being proportional to posterior probability
    '''
    post = U18_posterior(g,a,b,data); # calculate posterior given training data
        and prior parameters
    post = normalize_log_distribution(post) # normalize --> probability
        distribution

    # make a grid of all possible p_AdjN, p_NumN combinations at the granularity
        specified
    grid_adj = []
    grid_num = []
    for p_a in range(len(possible_p)):
        for p_n in range(len(possible_p)):
            grid_adj.append(possible_p[p_a])
            grid_num.append(possible_p[p_n])
    grid = zip(grid_adj,grid_num) # combine them because posterior probability is
        for the combination

    # samples some grammars!
    grammars = []
    for i in range(0,num_samps):
        r=choice(a=range(0,len(grid)),p=post)   # choose an index from the grid
            according to it's posterior probability
        grammars.append(grid[r])
    return grammars
```

→ *What does the `choice` function do? Hint: it's similar to the function `random.random()`, which generates a random number, but takes two arguments.*

Questions

1. The plotting function below takes sampled grammars from each of the four experimental conditions and plots them in the 2D grammar space.

```
import pylab as plt

def plot_grammars(g_1,g_2,g_3,g_4):
    '''
    Plot grammars sampled from posterior
    (results of multiple U18_roulette_wheel() calls)
    '''
    col = []
    for i in range(0,4):
        col.append([i for g in range(0,len(g_1))])

    plt.title("Sampled grammars")
    plt.xlabel("P(Num-N)");plt.ylabel("P(Adj-N)")
    plt.xlim(0,1);plt.ylim(0,1)

    x = [g[1] for g in g_1] + [g[1] for g in g_2] + [g[1] for g in g_3] + [g[1]
        for g in g_4]
    y = [g[0] for g in g_1] + [g[0] for g in g_2] + [g[0] for g in g_3] + [g[0]
        for g in g_4]

    plt.scatter(x,y,c=col)
    plt.show()
```

   Generate samples `g_1`,`g_2`,`g_3`,`g_4` using function calls like:

```
g_1=U18_roulette_wheel(g=fit_parameters[0],a=fit_parameters[1],b=fit_parameters
    [2],data=training_data[0],num_samps=100)
```

   Then call the plotting function to plot them. Does your plot look like Fig. 5?

2. What does it mean that the highest value of g is for component 1? Why do you think that might be the highest given the population of learners tested in the experiment? Come up with a new set of g values that you think would more accurately reflect the typology (e.g., in Fig. 1) and redo the samples and plot. Did it turn out as you expected?

3. What do you think would happen if the regularization bias were not as strong? Change the a,b parameters and see if you were right.

Extra credit: Iterating the model

The remaining functions in `U18_bayes.py` are for iterating the model–taking the original training data, generating posterior probability distributions for all four conditions, sampling some grammars from each posterior and counting the number of each pattern type that results. Then, generating some new training data from one of the sampled grammars from each condition to pass on to the next generation. And so on.

The first function classifies a given grammar as one of the four patterns. So if a learner acquires (0.7,0.8) that would be classified as a pattern 1 grammar. We'll need this to track the counts of each pattern type over generations.

```
def U18_classify(p_AdjN,p_NumN):
    '''Returns pattern type given (p_AdjN,p_NumN) pair.
    '''
    if p_AdjN > 0.5 and p_NumN > 0.5: return 1
    if p_AdjN < 0.5 and p_NumN < 0.5: return 2
    if p_AdjN < 0.5 and p_NumN > 0.5: return 3
    if p_AdjN > 0.5 and p_NumN < 0.5: return 4

    else: return 0
```

The second function generates training data given a grammar by taking sampled counts from a binomial distribution. We'll need this to create new training data to pass on to the next generation of learners.

```
def U18_produce(p_AdjN,p_NumN):
    '''Returns counts of Adj-N,N-Adj,Num-N,N-Num given (p_AdjN,p_NumN) pair.
    '''
    counts=[]
    AdjN = binomial(n=40,p=p_AdjN) # number of Adj-N out of n trials with p=p_AdjN
    NumN = binomial(n=40,p=p_NumN) # number of Num-N out of n trials with p=p_NumN
    counts.extend([AdjN, 40-AdjN, NumN, 40-NumN])
    return counts
```

And finally, a pretty horrendous looking function does the iterating.

```
def U18_iterate(starting_data,g,a,b,generations,num_samps):
    '''Iterates from starting data consisting of counts of Adj-N, Num-N
    Returns number of each pattern type left out of num_samps*4 in each generation
        .
    Steps:
    (1) get samples from each condition given starting_data
    (2) pick random grammar from each set of samples and use to generate new
        starting_data for that condition
    (3) count the number of each patterns resulting from those samples
    (4) REPEAT
    '''

    pattern_tracer=[[],[],[],[]] # value of the function, each sublist tracks
        count of each pattern type over generations

    for gen in range(0,generations):
        patterns_g=[] # accumulator for pattern types in each sample for the
            current generation
        new_data=[[],[],[],[]] # to be used as starting_data in the subsequent
            generation

        # for each condition, get sample of grammars, generate new training data,
            count patterns...
        for i in range(0,4):
            samps_i = U18_roulette_wheel(g,a,b,data=starting_data[i],num_samps=
                num_samps) # get sample of grammars for current condition
            r = random.randint(0,num_samps-1)# pick a random index from samps
            training_g = samps_i[r] # get the grammar at index r
            new_data[i] = U18_produce(training_g[0],training_g[1]) # use grammar
                to generate new starting data for current condition
            # now for each grammar in the sample, classify it and add pattern to
                the accumulator
            for s in range(0,len(samps_i)):
                patterns_g.append(U18_classify(samps_i[s][0],samps_i[s][1]))
```

→ *This function returns a list of lists. Each list in there tracks the counts for one of the four patterns over generations. If you ran 3 generations there'd be three numbers in each list representing how many of each pattern resulted from learning in that generation.*

→ *There are four loops. The outermost loop goes through the generations (as many as the user specifies).*

The second loop goes through each condition for the current generation, samples some grammars from the posterior for that condition and picks one of them to generate new training data from, to pass on to the next generation.

The third loop classifies all the sampled patterns.

The fourth loop puts the counts of each pattern for the current generation into the value of the function to be output.

```
        # go through pattern accumulator and count each type for the current
            generation
        for i in range(0,4):
            pattern_tracer[i].append(patterns_g.count(i+1)) # add the set of
                patterns for current condition to list

        starting_data=new_data # make the new training data the starting data

    return pattern_tracer
```

To do some iterating, call the function with the following arguments. It will take awhile, and you'll notice that the function you have in `U18_bayes.py` has some print statements so you know what it's up to while you're waiting.

```
counts=U18_iterate(starting_data=training_data,g=fit_parameters[0],a=
    fit_parameters[1],b=fit_parameters[2],generations=3,num_samps=100)
```

After that's done, use the plotting function below to see which patterns stick around and which die out over the generations. Do you think this is a realistic result? Why or why not?

```
def plot_counts(counts,generations):
    '''
    Plot count of grammars over generations (output of U18_iterate() call)
    '''
    m = (max(counts[0]),max(counts[1]),max(counts[2]),max(counts[3]))
    lim = max(m)+50

    plt.title("Counts of pattern type over generations")
    plt.xlim(1,generations);plt.ylim(0,lim)
    plt.xlabel("Generation");plt.ylabel("Count")
    plt.xticks(range(1,generations+1))

    for i in range(0,4):
        plt.plot(range(1,generations+1),counts[i],label='pattern '+str(i+1))
    plt.legend(loc='upper left',fontsize=12)
    plt.show()
```