# Simulating Language: Lab 10 Worksheet

Download bayes1.py from the usual place. This simulation implements a simplified version of the two-language model from Griffiths & Kalish (2007) using an explicit agent-based simulation - the original paper uses a slightly different method to reach the same conclusions.

The model of language in this simulation is much more abstract that what we have seen before. We are going to assume that there are just two language types, which we will call language 0 and language 1. You could think of these as two contrasting classes of language: maybe VO and OV languages, or languages which allow null subjects and languages which don't. Next week we'll look at a slightly more complex model where we explicitly model multiple language per type, but for the moment we'll simplify. Agents will produce (and learn from) data, which will simply exemplify which language type they use: agents who have acquired language type 0 will produce type 0 utterances (with some possibility of noise on transmission meaning they produce type 1 utterances instead), and vice versa.

Within this broad characterisation, we define a *prior bias* - a preference the learners have for one language type over the other. We implement *bayesian learning*, and experiment with the MAP or Sampling method (described below) for selecting a language from the posterior distribution.

As usual, the new code starts with a set of parameter declarations:

```
import random
import matplotlib.pyplot as plt

learning = 'sample'  # The type of learning ('map' or 'sample')
bias = 0.6           # The preference for language 1
noise = 0.2          # The probability of producing the wrong utterance
```

### Production of data

The function `produce` takes a language and produces an appropriate utterance. The function `generate_data` takes a language and a number, *n*, and produces a list of *n* utterances generated from the language.

```
# Produces an utterance for a particular language
def produce(language):
    if random.random() > noise:
        return language
    else:
        if language == 0:
            return 1
        if language == 1:
            return 0

# Generate a list of n utterances from a language
def generate_data(language, n):
    data_accumulator = []
    for i in range(n):
        utterance = produce(language)
        data_accumulator.append(utterance)
    return data_accumulator
```

*Can you see how 'noise' - errors on production - works?*

## The Bayesian bits

Recall that Bayes' rule allows us to calculate the relative *posterior* probability (the probability of each language given the data heard) from the *likelihood* (the probability of the data given each language) and the *prior* (the probability of each language, independent of the data - in other words, the learning bias).

The function `prior` returns the prior of a particular language.

```
# Gives the prior bias for a particular language.
def prior(language):
    if language == 1:
        return bias
    else:
        return (1 - bias)
```

- *If `bias` is over 0.5, which language has higher prior probability? If `bias` is under 0.5, which language has higher prior probability? What does it mean if `bias` is exactly 0.5?*

The function `likelihood` takes a language and a list of data and works out the likelihood of the data given the language.

```
# Calculates P(data | language)
def likelihood(data, language):
    total = 1.
    for utterance in data:
        if  utterance == language:
            total = total * (1. - noise)
        else:
            total = total * noise
    return total
```

- *Try it out with a particular language and a list of utterances. Does it give the numbers you are expecting?*
- *What role is noise playing in the calculation of likelihood? What would happen if there was no noise (i.e. noise=0.0)?*

## Learning

Bayesian learners calculate the posterior probability of each language based on some data, then select a language ('learn') based on those posterior probabilities. `select_language` implements this.

There are in fact two ways you could select a language based on the posterior probability distribution:

1. You could pick the best language - i.e. the language with the highest posterior probability. This is called MAP ("**m**aximum **a p**osteriori") learning.
2. Alternatively, you could pick a language probabilistically based on its posterior probability, without necessarily going for the best one every time (e.g. if language 0 has twice the posterior probability of language 1, you are twice as likely to pick it). This is called sampling (for "sampling from the posterior distribution").

The next bit of code implements both these ways of learning, using the familiar `wta` function to do MAP learning and using `roulette_wheel` to do sampling (compare the `roulette_wheel` function with the selection method from our evolutionary simulations early in the course).

```
# Picks a language give the posterior probabilities of all languages.
# This will either be the maximum a posteriori language ('map')
# or a language sampled from the posterior
def select_language(data):
    list_of_all_languages = [0,1]
    list_of_posteriors = []
    for language in list_of_all_languages:
        this_language_posterior = likelihood(data,language) * prior(language)
        list_of_posteriors.append(this_language_posterior)
    if learning == 'map':
        map_language = wta(list_of_posteriors)
        return map_language
    if learning == 'sample':
        sampled_language = roulette_wheel(list_of_posteriors)
        return sampled_language

# good old winner-take-all
def wta(items):
    maxweight = max(items)
    candidates = []
    for i in range(len(items)):
        if items[i] == maxweight:
            candidates.append(i)
    return random.choice(candidates)

# Given a list of scores, returns a position in that list selected randomly
# in proportion to its score
def roulette_wheel(scores):
    summed_scores = sum(scores)
    r = random.uniform(0,summed_scores)
    accumulator = 0
    for i in range(len(scores)):
        accumulator += scores[i]
        if r < accumulator:
            return i
```

- *Try out both ways of learning with data generated using the* `generate_data` *function we looked at earlier. Try and see when it will learn the language that generates the data correctly, and when it will fail to learn. Note that, for sampling learning, you may have to run* `select_language` *a few times for each set of data to get a sense of what it's doing.*

- *The function calculates the value of a variable called* `this_language_posterior` *for each language, but this isn't actually the real posterior probability. What's missing from Bayes' rule here, and why doesn't it matter?*

### The simulation

There are two main functions to actually carry out the relevant simulation runs. The first is `simulation`, which runs a single chain. It takes three parameters: the number of generations, the size of the bottleneck (i.e. the number of utterances the learner hears) and the frequency with which it should calculate statistics.

```
# Run a single chain for a particular bottleneck.
# Chains are initialised with a random initial language.
# Returns final language in chain, plus list detailing cumulative
# proportion of language 1 over time
def simulation(generations, bottleneck, output_every):
    language=random.randint(0,1)
    language1_count = 0.
    data_accumulator = []
    for i in range(1,generations+1):
        print language,
        language1_count+=language
        if output_every != 0:
            if (i % output_every) == 0:
                    data_accumulator.append(language1_count/i)
        data = generate_data(language, bottleneck)
        language = select_language(data)
    return [language,data_accumulator]
```

It returns a list of two things: the final language from the chain (needed by the `simulation_batch` function), and a (plottable) list of the proportion of generations which used language 1, calculated over the entire length of the chain to that point. It also prints out the language at each generation, so you can get a feel of how well each agent learns the language of its predecessor - you can suppress this behaviour by commenting out the print statement.

- *Try this out. What determines how quickly the language changes? What determines the language at the end of a chain?*

The final function is `simulation_batch`, which simply runs `simulation` over and over again and reports the proportion of those runs which ended up on language 1. It takes three parameters: the number of generations in each chain, the bottleneck, and how many simulations to run.

```
# Run a lot of simulations and returns the distribution of final languages
def simulation_batch(generations, bottleneck, number_of_runs):
    data_accumulator = []
    for i in range(number_of_runs):
        final_language=simulation(generations, bottleneck, 0)[0]
        data_accumulator.append(final_language)
    proportion_language1 = sum(data_accumulator)/float(number_of_runs)
    return proportion_language1
```

The distribution you get out of this can be thought of as equivalent to the set of cross-linguistic universals that we see in the world's languages today, assuming we're seeing the end of many many diffusion chains of cultural transmission that have been running for many generations.

## Questions

**Note:** Running the simulations takes some time, particularly if you run large batches using `simulation_batch`. You **definitely** want to comment out the print statement in simulation before running `simulation_batch` (i.e., put a "#" sign at the start of line 96). You may have to play with the parameters carefully to get reasonable results in a sensible time (remember, you can always use CTRL-C to cancel a run). To start with, try 100 runs for 100 generations each, and then to get cleaner results, try 1000 runs for 100 generations. For some parameter settings, you may need more generations, and higher bottleneck values will take longer to run. In general, you probably want to keep the bottleneck values between 1 and 10.

1. Griffiths & Kalish (2007) claim that the end result of iterated learning reflects precisely the bias of the learners, irrespective of the bottleneck size. This results holds for sampling learners. Can you replicate this result?
2. What happens if you switch from samplers to learners who select the MAP hypothesis - does the Griffiths & Kalish result still hold? (Note, because of the way this model has been simplified, you should avoid odd-numbered bottleneck sizes with MAP learners. We'll resolve this issue in the next lab.)
3. Finally, an implicit theme earlier in the course was that cultural evolution can take a very weak learning bias and amplify it into a strong effect. Can you demonstrate this using the simulation?