Simulating Language: Lab 10 Worksheet

Download `bayes2.py` from the usual place. This simulation implements a simplified version of the language model from Kirby, Dowman & Griffiths (2007) using an explicit agent-based simulation, and embeds this language model in a slightly more sophisticated population model.

Rather than using the two-grammar model we used in `bayes1.py`, we assume a language is made up of a set of *variables*, each of which can exist in a number of different *variant* forms. This is a rather general characterisation that actually applies well to a number of linguistic phenomena. For example, we can think of the variables as different syntactic categories, and the variants as word orders. Alternatively, the variables could be verb-meanings and the variants different realisations of the past tense, and so on. Agents will produce (and learn from) data which simply exemplifies which variant they have for a particular variable (with the possibility of noise on transmission). We will group languages into two classes: regular languages (where the same variant is used for all variables) and irregular languages (where more than one variant is used). In this respect the model is similar to the two-grammar model - we have two classes of languages - but there are now multiple languages per class.

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 regular languages
variables = 2            # The number of different variables in the language
variants = 2             # The number of different variants each variable can take
noise = 0.05             # The probability of producing the wrong variant
population_size = 1000   # Size of population
teachers = 'single'      # Either 'single' or 'multiple'
method = 'chain'         # Either 'chain' or 'replacement'
```

Production of data

The function `produce` takes a language, selects a random variant, and produces the relevant variant from the language.

```
# Produces a variant for a particular language and random variable
def produce(language):
    variable = random.randrange(len(language))
    correct_variant = language[variable]
    if random.random() > noise:
        return [variable,correct_variant]
    else:
        possible_noise_variants = range(variants)
        possible_noise_variants.remove(correct_variant)
        noisy_variant = random.choice(possible_noise_variants)
        return [variable,noisy_variant]
```

- *By looking at this code, can you tell how languages are represented in the simulation?*
- *Can you see how 'noise' - errors on production - works?*

Classifying languages

In this language model, prior probability is determined by language class: regular languages differ from irregular languages in their prior probability, and ultimately we are interested in the proportion of our simulated population who use regular languages. We therefore need a function to take a language and classify it as regular or not - the function regular does this.

```
# classifies a language as either regular (all variables expressed with the
# same variant) or irregular (multiple variants used)
def regular(language):
    regular = True
    first_variant = language[0]
    for variant in language:
        if variant != first_variant:
            regular = False
    return regular
```

The Bayesian bits

The function `prior` returns the prior of a particular language - if bias is over 0.5, regular languages have higher prior probability.

```
# Gives the prior bias for a particular language. Note that this must sum to
# 1 for all languages, so there is some normalisation in here
def prior(language):
    if regular(language):
        number_of_regular_languages = variants
        return bias / number_of_regular_languages
    else:
        number_of_irregular_languages = pow(variants, variables) - variants
        return (1 - bias) / number_of_irregular_languages
```

- *Why are we dividing the bias by the number of regular and irregular languages in this function? Check you understand how these numbers are calculated.*
- *How does this function differ from the prior from the Kirby, Dowman & Griffiths (2007) paper? (Hint: consider the case of more than two variables.)*

The function `likelihood` takes a language and a list of data and works out the likelihood of the data given the language - this is essentially the same function as in the two-grammar model, but modified to account for the fact that each utterance consists of a meaning and a form.

```
# Calculates P(data | language)
def likelihood(data, language):
    total = 1.
    for utterance in data:
        variable = utterance[0]
        variant = utterance[1]
        if variant == language[variable]:
            total = total * (1. - noise)
        else:
            total = total * (noise / (variants - 1))
    return total
```

Learning

Bayesian learners calculate the posterior probability of every possible language based on some data, then select a language ('learn') based on those posterior probabilities. `all_languages` enumerates all possible languages using a cute recursive method (don't worry too much if you can't figure out how it works!), `select_language` implements hypothesis selection - this is essentially the same function as in `bayes1.py`, so I have eliminated `wta` and `roulette_wheel` here to save space.

```
# Returns a list of all possible languages for expressing n variables
def all_languages(n):
    if n == 0:
        return [[]]
    else:
        result = []
        smaller_langs = all_languages(n - 1)
        for l in smaller_langs:
            for v in range(variants):
                result.append(l + [v])
        return result

# 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 = all_languages(variables)
    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_index = wta(list_of_posteriors)
        map_language = list_of_all_languages[map_language_index]
        return map_language
    if learning == 'sample':
        sampled_language_index = roulette_wheel(list_of_posteriors)
        sampled_language = list_of_all_languages[sampled_language_index]
        return sampled_language
```

The simulation

There are two main functions to actually carry out the relevant simulation runs. The first is `pop_learn`, creates a new population of a specified size who learn a language from data produced by an adult population. It calls on the `teachers` global parameter to decide whether these learners should earn from a single individual in the adult population, or whether it learns each utterance from a randomly-selected member of the adult population (i.e. learns from multiple teachers).

- *How is the difference between single and multiple teachers implemented? In the multiple-teacher version, is each data item guaranteed to be produced by a separate teacher?*

```
# Generates a new population, consisting of a specified number_of_learners,
# who learn from data generated by the adult population
def pop_learn(adult_population,bottleneck,number_of_learners):
    new_population = []
    for n in range(number_of_learners):
        if teachers == 'single':
            potential_teachers = [random.choice(adult_population)]
        if teachers == 'multiple':
            potential_teachers = adult_population
        data = []
        for n in range(bottleneck):
            teacher = random.choice(potential_teachers)
            utterance = produce(teacher)
            data.append(utterance)
        learner_grammar = select_language(data)
        new_population.append(learner_grammar)
```

`initial_population` is a subsidiary function which generates a population of a specified size of individuals speaking randomly-selected languages.

```
    return new_population
# Returns a list of n randomly languages
def initial_population(n):
    population = []
    possible_languages = all_languages(variables)
    for agent in range(n):
        language=random.choice(possible_languages)
        population.append(language)
    return population
```

The second main function is `simulation`, which is the top-level function which actually runs simulations. This function calls on the `method` parameter, to run either `chain` simulations (where a population consists of a series of generations, where the entire population is replaced at each generation) and `replacement` simulation (where a single individual is replaced at each 'generation'). It returns a list of two things: the final population, and a (plottable) list of the proportion of generations which used regular languages.

```
# Returns a list of two elements: final population, and accumulated
# data, which is expressed in temrs of proportion of the population using
# a regular language
def simulation(generations, bottleneck, report_every):
    population = initial_population(population_size)
    data_accumulator=[]
    for i in range(1,generations+1):
        if method == 'chain': # Replace whole population
            population = pop_learn(population, bottleneck, population_size)
        if method == 'replacement': #Replace one individual at a time
            population = population[1:]
            new_agent = pop_learn(population, bottleneck, 1)[0]
            population.append(new_agent)
        if (i % report_every == 0):
            regular_language_count = 0
            for agent in population:
                if regular(agent):
                    regular_language_count += 1
            data_accumulator.append(regular_language_count / float(population_size))
    return [population,data_accumulator]
```

Questions

Note: As for `bayes1.py`, running the simulations takes some time, particularly if you run large populations for large numbers of generations. In general, you probably want to keep the bottleneck values between 1 and 10, in which case you should get representative results within 100 to 500 generations (for chain populations). Larger populations (e.g. 1000 individuals) generally give you cleaner results (have a think about why this is).

1. Using the default parameters (single teacher, chain method), check that you can replicate the standard results for sampling and MAP learners: convergence to the prior for samplers, exaggeration of the prior for MAP. Also verify that the annoying odd vs. even bottleneck result for map learners goes away now we have dropped the two-grammar model.
2. What happens if you switch from single teachers to multiple teachers? Does the sampler result change? Does the MAP result change? How does the bottleneck effect these results? Is this what you expected?
3. Finally, what happens if you switch from the chain method to the replacement method? Don't forget that each 'generation' in a replacement simulation just replaces a single individual, so you'll have to run the simulations for lots more generations to get equivalent results to those you got under the chain method.