ABCM_Take-home_assignment_s1099392

November 6, 2022

1 Take-home assignment: Agent-based Cognitive Modelling

Please complete each of the conceptual questions in a markdown cell (in written text), and each of the coding questions using code cells (in combination with markdown cells if a written part is also required for answering the question).

It is important that you complete each of the exercises in this take-home assignment **individually**. If we see signs of answers being shared between students, we will investigate.

Exercise 1 (Conceptual question):

This question is about agent-based cognitive modelling in general.

Below are four research questions. For each of these, write down: - Whether or not you think *agent-based* modelling would be a sensible approach to address that research question, and explain why. - If your answer is that agent-based modelling would be a sensible approach, also write down whether the agents in this model would have to be *cognitive* agents, and explain why.

1. Is categorisation in humans exemplar-based or feature-based? (In simple terms, *exemplar-based* means: if a novel stimulus is similar to other dogs I've seen, it's probably a dog. While *feature-based* means: if it barks, has four legs, and a wagging tail, it's probably a dog.)
2. Do more extreme ideas spread through a population more quickly than more moderate ideas?
3. When two individuals do a task like moving a sofa together (i.e., *joint action*), how do they coordinate?
4. How does eye colour spread through a population? (Assuming, for example, that the allele for blue eyes is recessive and the allele for brown eyes is dominant.)

1. I don't believe agent-based modelling would be a sensible approach to address the research question because the research question concerns cognitive processes within individual rather than interaction between agents.
2. Agent-based modeling would be a sensible approach to address this research question because the spead of ideas through a population can only be done via interactions between agents in the population, which is of primary concern of an agent-based model. The agents in the model should be cognitive agents, as the speed at which extreme/moderate ideas spread would be influeneced by which ideas are more likely to be retained by individuals and to be brought up in a conversation. How likely an idea is to be brought up in a conversation depends on how much one agent thinks: (1) the idea is interesting/worth mentioning and (2) the other agent will be interested in the topic, which both require agents to be capable of cognitive processes.

3. Agent-based modeling would be a sensible approach to address this research question because the research question focuses on how two individuals model the partner's mental state to successfully coordinate in their movements (e.g., how high should one hold the sofa). This is also a reason why the agents should be cognitive agents to address this research question.

4. Agent-based modeling might not be a sensible approach to address this research question if the spread of eye color is only determined by genetics. However, agent-based modeling could be a sensible approach if, for example, people have cognitive biases to find someone with blue eyes more attractive than someone with brown eyes because it might have some impact on how people choose their mates, which could also have impact on how eye color spread through a population.

Exercise 2 (Conceptual question):

This question is about game theory and pay-off matrices as a representation of social coordination situations.

Imagine the following situation: - An employer has to make a decision about whether to pay their employee a low or a high salary, without knowing how much effort the employee is going to put into their work. If the employee puts in a high amount of effort, the high salary is worth it. If, instead, the employee puts in a low amount of effort, it's better (from the employer's point of view) to pay them a low salary. - Simultaneously, the employee has to make a decision about whether to put high effort or low effort into their work, without knowing how much the employer is going to pay them. If the employer decides to pay them a high salary, the high amount of effort is worth it. If, instead, the employer decides to pay them a low salary, it's better (from the employee's point of view) to put in a low amount of effort.

Below is an empty pay-off matrix (game-theory style). Translate the situation above to a pay-off matrix, by filling in each of the cells in the table below, according to the situation described above. Replace each of the A's and B's in the table with the pay-off values for player A (the employer) and Player B (the employee), using the following values: $[0, 1, 2, 3]$. Also write out and explain your considerations that went into deciding which numbers to put in each cell of the pay-off matrix.

B (Employee):	High effort	Low effort
A (Employer):		
High salary	2, 2	0, 3
Low salary	3, 0	1, 1

I put the highest number (3) for cases where the employer (A) pays low salary and the employee (B) put high effort (i.e., the employer gets high reward [workload done by the employee] with low sacrifice [salary]) and where the employee (B) gets high salary with low effort. In other words, these are two situations that are optimal for one of the agent but are the worst for the other agent. When A pays high salary and B puts high effort, they are both happy because each agent gets what they want (win-win). However, this is not as the most ideal senario for each agent, as getting high reward without losing much is better than getting high reward with losing much. This is why I put 2 for high salary / high effort mattrix. Lastly, I put 1 for low salary / low effort mattrix because neither agent is winning: the employer (A) gets less workload done by the employee (B), and B gets low salary. However, this is still better than paying high salary for low workload or getting low salary for high effort, so the value should be higher than 0 but less than 2.

Exercise 3 (Coding question):

Use the tomsup package to simulate the following situation:

- agent0 = A `'1-ToM'` agent with default parameter settings
- agent1 = A `'2-ToM'` agent with default parameter settings
- game = `'party'`
- environment = `'round-robin'`
- n_sim = 10

Run 10 simulations of this interaction for a number of rounds that seems reasonable to you, and use the `group.plot_p_k()` method to plot how agent1's belief about agent0's ToM level changes over time. (The three code cells below make a start by loading in the relevant packages.)

Find out whether there is a certain number of rounds after which each of the 10 simulations reaches a point where agent1 has a fully accurate model of their opponent's k-level (and has reached maximum certainty about that).

Show a plot to back up your answer, and also explain your answer fully in words.

[ ]:
```
#!pip install tomsup
```

[ ]:
```
import tomsup as ts
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
party = ts.PayoffMatrix(name='party')

print(party)
```

```
<Class PayoffMatrix, Name = party>
The payoff matrix of agent 0
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  5 |  0 |
agent 0|  1 |  0 | 10 |

The payoff matrix of agent 1
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  5 |  0 |
agent 0|  1 |  0 | 10 |
```

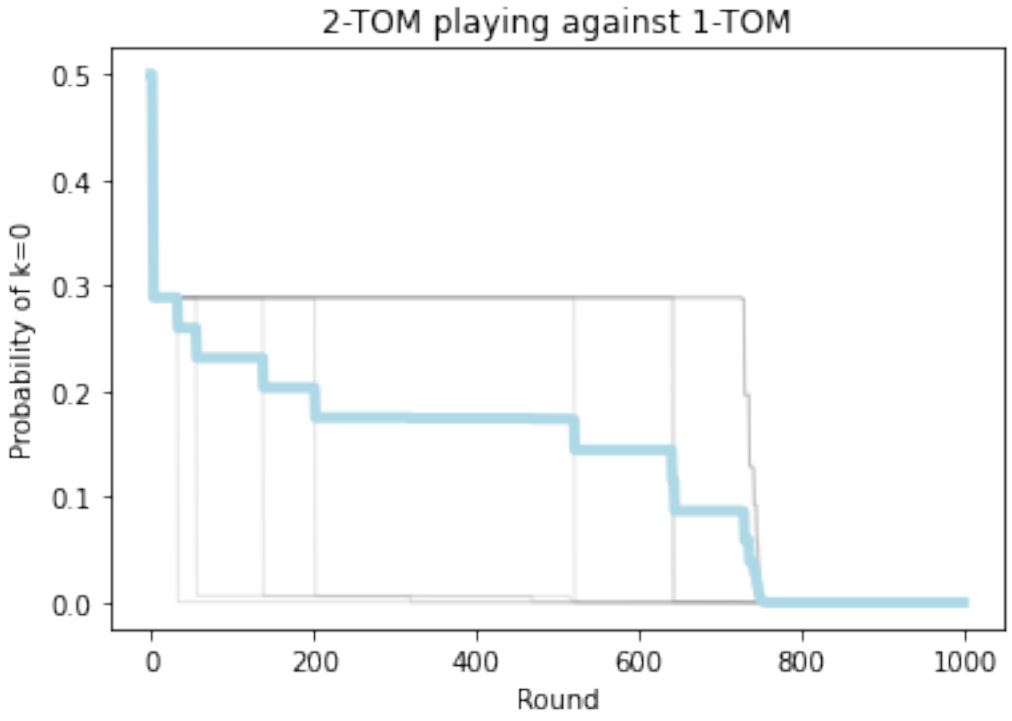
[ ]:
```
agent_types = ['1-TOM', '2-TOM']
starting_parameters = [{}, {}]

group = ts.create_agents(agent_types, starting_parameters)
```

```
group.set_env(env='round_robin')
```

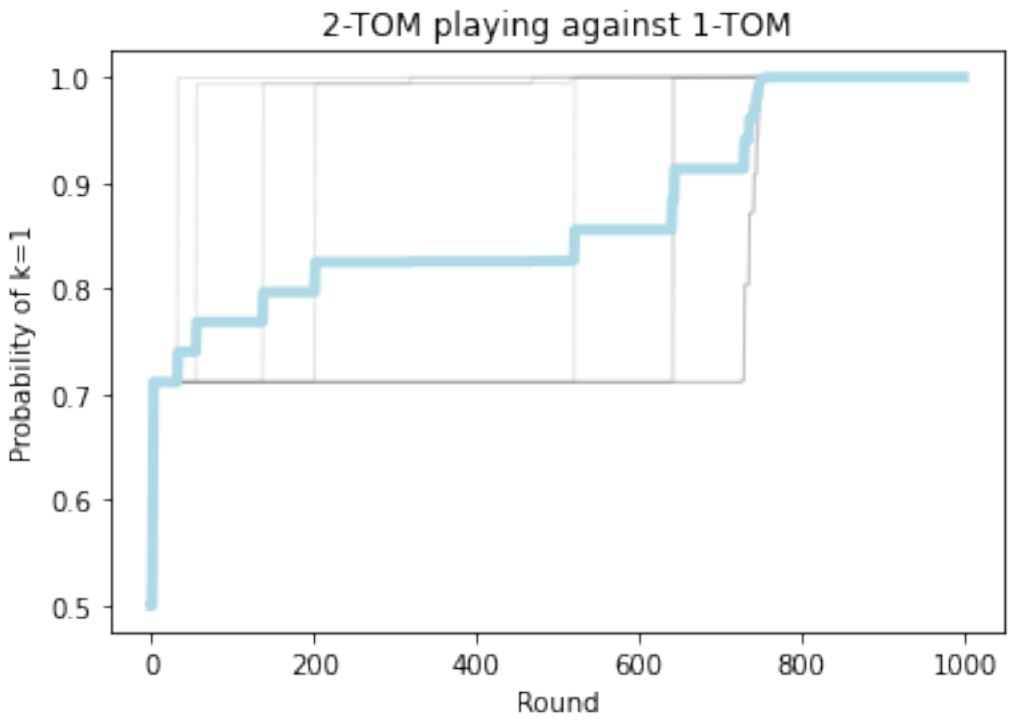
[ ]:
```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning) #Suppress future␣
 ↪warnings

results = group.compete(p_matrix='party', n_rounds=1000, n_sim=10,␣
 ↪save_history=True, verbose=False)
```

[ ]:
```
#agent 1's estimation about the probability of agent 0 being 0-ToM agent
group.plot_p_k(agent0="1-TOM", agent1="2-TOM", agent=1, level=0)
#agent 1's estimation about the probability of agent 0 being 1-ToM agent
group.plot_p_k(agent0="1-TOM", agent1="2-TOM", agent=1, level=1)
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



The figures show that when n_sim is 10, approximately 750 rounds are required for agent1 to have an accurate model of agent0's k-level because around 750 the probability of agent0 (1-ToM agent) being 0-ToM agent becomes 0 and that of agent0 being 1-ToM agent becomes 1.

Exercise 4 (Conceptual question):

This question is about both Waade et al. (2022) and de Weerd et al. (2015), and the comparison between these two models.

In both the Waade et al. (2022) model and the de Weerd et al. (2015) model, the k-ToM agent has a belief about the k-level of the agent they're interacting with, and updates this belief over the course of the interactions. Describe the main similarities and differences in how this belief-updating about the other agent's k-level works in these two different models.

The main similarity between Waade et al. (2022) and Weerd et al. (2015) in how the belief-updating about the partner's k-level is done is that both models update the belief by learning how the agent they are interacting with behaves based on their previous behavior and computing the probability of the partner being k-level agent based on the learning. Another similarity is that in both models, a k-ToM agent can estimate their partner to have up to (k-1)-ToM level (e.g., 2-ToM agent do not estimate the other agent to be 2-ToM agent but either 1-ToM or 0-ToM agent, depending on the model). The main difference is that while in the Waade et al. (2022) model a k-ToM agent estimates their partner to be either (k-1)-ToM agent or less (e.g., 2-ToM agent estimates the other to be either 1-ToM or 0-ToM agent but NOT 2-ToM agent), in the Weerd et al. (2015) model, a k-ToM agent always assumes their partner to be (k-1)-ToM agent (2-ToM agent always assumes that the agent they are interacting with is a 1-ToM agent).

Exercise 5 (Conceptual question):

This question is about de Weerd et al. (2015).

Imagine you want to know whether actual humans adapt their strategy in the Tacit Communication Game depending on their estimate of the level of ToM that their interlocutor uses. Imagine that your experiment consists of the following two conditions (where in both conditions, the participant is *told* that they are playing the game with another participant, even though in reality, their interlocutor is being simulated by a computer): - In the *0-ToM* condition, participants interact with a computer that is implemented as a zero-order ToM agent from the de Weerd et al. (2015) model. - In the *1-ToM* condition, participants interact with a computer that is implemented as a 1st-order ToM agent from the de Weerd et al. (2015) model.

Now imagine that you have collected your experimental data and you are ready to analyse your results. These are the specific hypotheses you would like to test:

- *Null hypothesis*: Human participants do not adapt their communication strategy to the ToM-level of their interlocutor, and will always behave like a 2-ToM sender.
- *Alternative hypothesis*: Human participants *do* adapt their communication strategy to the ToM-level of their interlocutor. Specifically, participants in the *0-ToM* condition will behave more like a *1-ToM* sender, and human participants in the *1-ToM* condition will behave more like a *2-ToM* sender.

Explain how you could use the model described in de Weerd et al. (2015) to test these hypotheses, *using* the data you've collected in your experiment as described above. Describe the process step-by-step.

Although I am not sure if the data contains beliefs about the possible interpretation by the interlocutor for each k-level (like shown in the demonstration), I assume this is the case and construct my answer based on this assumption.

1. For each trial, extract (1) the move made by the human sender, (2) the receiver's goal, (3) the receiver's beliefs about the possible interpretation of the message made by the sender for each k-level, and (4) the condition the sender was assigned to.
2. Compare the receiver's goal with one of the receiver's beliefs for each k-level for the particular move made by sender, and make a new dataframe. For each k-level of beliefs, if the receiver's goal matches to one of the receiver's beliefs for the particular move, put k for "receiver_belief" and 1 for "correct" (e.g., if the receiver's goal matched to one of the receiver's beliefs for the particular move for 2-ToM sender but not 1-ToM, then the dataframe for the trial would look like the table below).

participant	*condition*	*trial_n*	*receiver_belief*	*correct*
1	0-ToM	1	1	0
1	0-ToM	1	2	1
2	1-ToM	1	1	0
2	1-ToM	1	2	1

3. Perform a (mixed-effects) logistic regression model with "correct" as the response variable and "receiver_belief" and "condition" as fixed effects. If the "receiver_belief", which codes the k-level of the sender on which the receiver's belief about possible goals is based, is a

statistically reliable factor and estimates the probability of the receiver making a correct interpretation about the sender's message is higher for 2-ToM than 1-ToM, then it suggests that the human sender is more likely to be using the second order theory of mind regardless of the receiver's k-level. However, if there is an significant interaction between "receiver_belief" and "condition", especially when the probability of "correct" being 1 is higher for 1-ToM sender in the 0-ToM receiver condition and is higher for 2-ToM sender in the 1-ToM receiver condition, then it suggests that the human sender is likely to adapt their communication strategy to the k-level of the receiver.

Exercise 6 (Implementation/code-related question):

This question is about Cuskley et al. (2018).

Imagine that you want to extend the Cuskley et al. (2018) model to look at the effect of social network structure on morphological complexity. To give you an idea of what this might look like, three different possible social network types are described at the bottom of this exercise (`"fully_connected"`, `small-world` and `"scale-free"`). Describe *in words* how you would have to adapt the code of Computer Lab 3 in order to be able to create populations with these three different social network types. More specifically, answer the three questions below:

a) How would you go about giving structure to the population (i.e., specifying which agent is connected with which other agents)? The code in Computer Lab 3 consists of several different classes, what attributes would you have to add to which of these classes in order to create such structure?

b) Assuming you have now added attributes to the relevant classes in order to specify for each agent to which other agents in the population it is connected. To which class would you have to add a method that can initialise a population with a specified type of social network structure (i.e., a method that takes the type of social network structure as one of its input arguments; an input argument that can be set to `"fully_connected"`, `small-world` or `"scale-free"`)?

c) Finally, once you've specified steps **a)** and **b)**, which method of which class would you have to adapt to make sure that agents only interact with agents they are connected to?

Fully connected network: This network is maximally dense, such that all possible connections are realized (i.e., all agents in the population get to interact with each other). It is also homogenous, in the sense that every agent has the same number of connections.

Small-world network: This network is also relatively homogenous such that every agent has approximately the same number of connections, yet it is much sparser than the fully connected network and realizes only half of the possible connections. This network type has the small-world property of "strangers" being indirectly linked by a short chain of individuals.

Scale-free network: This network is equally sparse as the small-world network, and it has the same number of possible connections overall. However, it is not homogenous: not every agent has the same number of connections. While some agents are highly connected, others are more isolated. The distribution of connections in this network roughly follows a power-law distribution, with few agents having many connections (forming "hubs" who interact with almost everyone in the population), and a tail of agents having very few connections.

a) I would add "network_structure" to the class Simulation in which each agent will be assigned a number that indicates how many agents they can be connected to. If the network parameter is set to the "fully_connected", each agent will be given [n - pop_size] so that they can be connected to

every single other agent in the population. If the network parameter is set to "small-world", then each agent will be given a fixed number that can range from 1 - 100. If the network parameter is set to "scale-free", then each agent will be given a random number from 1 - 100. After the number is assigned, each agent will be connected to other agents. The numver of agents each agent can be connected to is determined by the number assigned earlier. For the "fully_connected" network, each agent will be linked to every single agent in the population. For the "small-world" network, every agent will be linked to 1-100 agents in the population depending on the number assigned, and the number of agents each agent can be connected is the same for all agents. For the "scale-free" network, each agent will be linked to 1-100 agents in the population depending on the number assinged, and the number of agents each agent can be connected to is randomly chosen from 1 - 100 for each agent.

b) I would add a network initialization method to the class Simulation. The imput argument can be globally set like other parameters such as population size.

c) The timestep() method in Simulation needs to be adapted because this is where each agent is assigned a role of producer and receiver and initiate interactions.

Exercise 7 (Coding question):

This question is about Mudd et al. (2022).

Imagine that you'd want to adapt the model of Mudd et al. (2022) to change the way in which agents update their vocabulary when they find out that they are using two different forms for the same concept. Instead of only the receiver doing *bit update* to make their form more similar to that of the sender, you want to adapt the model such that *both* the sender and the receiver update their language representation by adapting to each other; specifically, by finding a middle ground between their two forms. (Like two speakers of English who start out with the two different forms *sofa* and *couch*, and adapt by both updating their form to a middle ground form like *souch*.)

Below is an empty skeleton of a function called `update_forms()`. Complete this function so that it finds a middle ground between the producer's form and the comprehender's form, and updates both the producer's and the comprehender's language representation with this new form. **Note** that the resulting form that the two agents update their language representation with should still (or again) be a vector containing bits (i.e., only 0s and 1s). (Because changing this to continuous values would require more extensive changes to the rest of the code and model.)

For context: this new `update_forms()` function would be replacing the `update_comprehender_concept()` function (final code cell in section 1.3 of Computer Lab 4).

[ ]:
```
# CODE SKELETON TO COMPLETE FOR EXERCISE 7:

# DEFINITIONS:
# producer.language_rep[producer_concept_choice][1] = producer's form for the current concept, as in Lab 4
# comprehender.language_rep[producer_concept_choice][1]) = comprehender's form for the current concept, as in Lab 4

def update_forms(producer, producer_concept_choice, comprehender):
```

```
    # Step 1: Find the middle ground between the producer's form and the␣
 ↪comprehender's form:
    new_form = []
    counter_correction = 0

    comparison_list = ([(p_bit == c_bit) for p_bit, c_bit in zip(producer.
 ↪language_rep[producer_concept_choice][1], comprehender.
 ↪language_rep[producer_concept_choice][1])])

    correctable_indexes = [i for i, comparison in enumerate(comparison_list) if␣
 ↪comparison == False]  # get False indeces
    for i, bit in enumerate(comprehender.
 ↪language_rep[producer_concept_choice][1]):
        #change the first n of comprehender's bits to the sender's bits
        #n = the number of correctable_indexes / 2 -> half of the mismatching␣
 ↪bits
        if counter_correction < len(correctable_indexes)/2:
            if i in correctable_indexes:
                bit = producer.language_rep[producer_concept_choice][1][i]
                counter_correction += 1
        new_form.append(bit)

    # NOTE: Before moving on to Step 2, make sure that the middle ground form␣
 ↪created above is
    # (or gets converted back into) a bit vector (i.e., containing only 0s and␣
 ↪1s)

    # Step 2: Update the producer's and comprehender's language representations␣
 ↪to the new form:
    producer.language_rep[producer_concept_choice][1] = new_form
    comprehender.language_rep[producer_concept_choice][1] = new_form

    pass
```

[ ]:
```
#The code below for checking if Step 1 is working as expected
producer_bit = [1,0,1,1,1,1]
receiver_bit = [1,0,1,0,1,0]

new_form = []
counter_correction = 0
comparison_list = ([(p_bit == c_bit) for p_bit, c_bit in zip(producer_bit,␣
 ↪receiver_bit)])

correctable_indexes = [i for i, comparison in enumerate(comparison_list) if␣
 ↪comparison == False]  # get False indeces
for i, bit in enumerate(receiver_bit):
```

```
    if counter_correction < len(correctable_indexes)/2:
        if i in correctable_indexes:
            bit = producer_bit[i]
            counter_correction += 1
    new_form.append(bit)
print(new_form)
```

```
[1, 0, 1, 1, 1, 0]
```

Exercise 8 (Model design question):

This question is about agent-based cognitive modelling in general.

Below is a research question, followed by a verbal explanation of what this research question is trying to get at. How would you design an agent-based model to answer this research question? (More specific instructions for what to specify follow below.)

Research question: What is more important for successful problem-solving: expertise or diversity?

Explanation of the question: Imagine a population in which individuals can develop different strategies for solving a particular problem. For ease of explanation, let's imagine the problem is something practical, and the solution is to design and build a particular tool that consists of different components. Imagine each component has a value that represents how much it contributes to solving the problem, and that an optimal solution to the problem requires a tool that combines several optimal components. Imagine individuals in this populations can have one of two possible strategies:

1. Select one individual from the population who you want to learn from, spend a lot of time to perfectly acquire their solution (i.e., how to make their variant of the tool), and innovate that variant.
2. Take in examples from many different individuals in the population, and try to combine their solutions (i.e., their variants of the tool).

This research question is getting at a trade-off between accuracy of learning and diversity of input. An important assumption of your model should be that each individual has the same limited amount of time. Spending more time on learning one variant perfectly (as in strategy 1) means that you will be able to reproduce the variant more accurately, and understand better how it works (which should allow you to make more targeted innovations), but it comes at the cost of not being able to see a diverse set of solutions to the problem. Vice versa, taking in examples from different individuals in the population (strategy 2) has as an advantage that you'll be able to take in a diverse set of possible solutions to the problem (which should allow you to combine the good parts of the different solutions), but that comes at the cost of not being able to acquire/reproduce these perfectly (because you can't spend as much time learning about each individual solution).

Specify, in bullet points, what the three major components of the model should consist of: - the agents - the interactions (agent-agent interactions and/or agent-environment interactions) - the environment

If you think one of these three components is not relevant for answering this research question, write "not relevant" and briefly explain *why* you believe this component is not relevant to the question.

- the agents: Individuals in the population who can develop different solutions for a problem. The agents can choose either to select one agent from the population to "discuss" a strategy

for solving a problem to innovate a solution or to interact with many agents to combine their solutions. This preference can be set as a parameter.

- the interactions: Communication between agents
- the environment: The time each individual has (although this could be part of agent's feature).

Exercise 9 (Coding question):

This question is about Mudd et al. (2022).

Mudd et al. (2022) find that population size has an effect on the degree of lexical variability in the population, where larger populations lead to less lexical variability (i.e., more convergence). Their explanation for this effect is that this is a result of the feedback loop illustrated in Figure 12 in the paper (copy-pasted below):

However, Mudd et al. (2022) do not show plots with the proportion of language game results to illustrate what this hypothesised feedback loop would look like in a single simulation. So that is what you are going to try and do below.

a) Imagine you run a simulation contrasting a population of 5 agents with a population of 100 agents, and these simulations would behave according to the feedback loop described in Figure 12 of Mudd et al. (2022). Now imagine you would generate plots of the proportion of language game results (similar to Figures 7 and 8 from the Mudd et al. paper) for each of these populations. Describe in words what you think these plots should look like to illustrate the feedback loop. How would you expect the plots for the small and large population to be different? Explain why.

b) Now actually run these simulations and generate the corresponding plots with the proportion of language game results, by adapting the final three code cells in this notebook. Instead of contrasting `n_groups` = 1 with `n_groups` = 10, your code should contrast `n_agents = 5` with `n_agents = 100`. Set the `n_groups` parameter to `n_groups = 5` (for both simulations). Run these simulations about 10 times, until you find an example case that looks like what you've described for part **a)** of this exercise. **Note** that this will definitely not happen every time you run the simulations and compare a single simulation with population size 5 with a single simulation with population size 100, but it will happen *sometimes* (maybe around 1/3 of the time). The difference does not have to be very stark, but it does have to be visible.

1.1 Necessary installations and imports:

[ ]:
```
#pip install mesa
```

[ ]:
```
import random
import numpy as np
import itertools
from math import sqrt
import time
from mesa import Agent, Model
from mesa.datacollection import DataCollector
from mesa.time import RandomActivation
from mesa.batchrunner import BatchRunner, FixedBatchRunner
import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns
```

1.2 Parameter settings:

[ ]:
```
################## PARAMETER SETTINGS: #################

test_params = dict(
    n_concepts=10, # int: number of concepts
    n_bits=10,  # int: number of bits (determining length of forms and␣
 ↪culturally-salient feature vectors)
    n_agents=10, # int: number of agents in the population
    n_groups=1,  # determines how many different semantic groups there are
    initial_degree_of_overlap=0.9,  # degree of overlap between the form and␣
 ↪meaning components
    n_steps=2000  # number of timesteps to run the simulation for (called␣
 ↪"model stages" in the paper)
)
```

1.3 Initialising the population and their language representations:

[ ]:
```
def language_skeleton(n_concepts, n_bits):
    """ initiate language with n_concepts and n_bits
    in the form {0: [meaning, form], 1: [meaning, from], ...}
    the meaning and form components are initiated with None """
    skeleton_concept_meaning_form = {}
    for n in range(n_concepts):
        skeleton_concept_meaning_form[n] = [[None] * n_bits] * 2
    return skeleton_concept_meaning_form
```

[ ]:
```
def language_create_meanings(n_concepts, n_bits, n_groups):
    """ generate the meaning representation for each group
    returns a dictionary with group: meaning representation
    ex. {0: [[1, 1, 0, 1, 1], [0, 0, 0, 1, 0]], 1: [[0, 0, 0, 1, 1], [1, 1, 1,␣
 ↪1, 0]]} """
    group_meaning_dic = {}
    for n in range(n_groups):
        condition = False
        while condition == False:
            single_group_meaning_list = []
            for concept in range(n_concepts):
                single_group_meaning_list.append(random.choices([0, 1],␣
 ↪k=n_bits))  # list of len n_components
            if len(set(tuple(row) for row in single_group_meaning_list)) == len(
                    single_group_meaning_list):
                condition = True
```

```
                group_meaning_dic[n] = single_group_meaning_list

    return group_meaning_dic
```

[ ]:
```
def language_add_meaning(agent, meaning_dic):
    """ takes in the language skeleton and adds the meaning component depending␣
 ↪on group of agent """
    counter = 0  # to keep track of which meaning component in meaning_dic␣
 ↪values
    for concept, meaning_form in agent.language_rep.items():
        meaning_form[0] = meaning_dic[agent.group][counter]  # meaning_form[0]␣
 ↪is the meaning only
        counter += 1
    return agent
```

[ ]:
```
def language_add_form(agent, initial_degree_of_overlap):
    """ start with meaning representation and assign form representation
    depending on the desired degree of overlap """
    for concept, meaning_form in agent.language_rep.items():
        forms = []
        for bit in meaning_form[0]:
            my_choice = np.random.choice([True, False],␣
 ↪p=[initial_degree_of_overlap, 1 - initial_degree_of_overlap])  # p = weights
            if not my_choice:  # if my_choice == False
                random_choice = np.random.choice([0, 1])
                forms.append(random_choice)  # random choice 0 or 1 if False␣
 ↪(random)
            else:
                forms.append(bit)  # append the same bit (iconic)
        meaning_form[1] = forms
    return agent
```

1.4 Running a language game and updating the agents' language representations:

[ ]:
```
def language_game(sorted_agent_list):
    """ takes agent list sorted by group
    chooses and agent to be the producer """
    form_success = 0
    meaning_success = 0
    bit_update = 0

    for a in sorted_agent_list:
        what_is_updated = language_game_structure(a, sorted_agent_list)

        if what_is_updated == "3a":
```

```
            form_success += 1
        elif what_is_updated == "3b1":
            meaning_success += 1
        else:  # "3b2"
            bit_update += 1

    language_game_stats = {"form_success": form_success, "meaning_success":␣
↪meaning_success, "bit_update": bit_update}
    return language_game_stats
```

[ ]:
```
def language_game_structure(producer, all_agents):
    comprehender = random.choice(all_agents)
    producer_concept_choice = random.choice(list(producer.language_rep))  # 1
    form_match_answer = does_closest_form_match(producer,␣
↪producer_concept_choice, comprehender)  # 2
    if form_match_answer == False:  # 3b
        meaning_match_answer = does_closest_meaning_match(producer,␣
↪producer_concept_choice, comprehender)
        if meaning_match_answer == False:  # 3b2
            update_comprehender_concept(producer, producer_concept_choice,␣
↪comprehender)
            return "3b2"
        else:  # 3b1
            # None
            return "3b1"
    else:  # 3a
        # None
        return "3a"
```

[ ]:
```
def does_closest_form_match(producer, producer_concept_choice, comprehender):
    produced_form = producer.language_rep[producer_concept_choice][1]

    distance_from_produced_form = {}
    for concept, meaning_form in comprehender.language_rep.items():
        # compare produced concept and all comp concepts, calculate distance␣
↪between each
        distance = sum([abs(prod_bit - comp_bit) for prod_bit, comp_bit in␣
↪zip(produced_form, meaning_form[1])])
        distance_from_produced_form[concept] = distance

    min_distance = min(distance_from_produced_form.values())
    comp_closest_form_list = [concept for concept, distance in␣
↪distance_from_produced_form.items() if distance == min_distance]
    comp_chosen_form = random.choice(comp_closest_form_list)  # because there␣
↪can be multiple, randomly choose from list
```

```
    return producer_concept_choice == comp_chosen_form  # returns True or False
```

[ ]:
```
def does_closest_meaning_match(producer, producer_concept_choice, comprehender):
    produced_form = producer.language_rep[producer_concept_choice][1]

    distance_from_produced_form = {}
    for concept, meaning_form in comprehender.language_rep.items():
        # compare produced concept and all comp concepts, calculate distance between each
        distance = sum([abs(prod_bit - comp_bit) for prod_bit, comp_bit in zip(produced_form, meaning_form[0])])
        distance_from_produced_form[concept] = distance
    comp_closest_meaning = min(distance_from_produced_form, key=distance_from_produced_form.get)

    return comp_closest_meaning == producer_concept_choice  # returns True or False
```

[ ]:
```
def update_comprehender_concept(producer, producer_concept_choice, comprehender):
    """ update comprehender form
    compare all producer and comprehender form, find the ones that don't match
    of the ones that don't match, choose one and flip this bit of the comprehender's form """
    comparison_list = ([(p_bit == c_bit) for p_bit, c_bit in zip(producer.language_rep[producer_concept_choice][1], comprehender.language_rep[producer_concept_choice][1])])
    # to prevent case where correct concept has a match for form producer and comprehender
    # this could happen if comprehender has 2 forms which both == form producer and the non-matching concept one gets chosen
    if all(comparison_list) == True:
        pass
    else:
        correctable_indexes = [i for i, comparison in enumerate(comparison_list) if comparison == False]  # get False indeces
        chosen_index_to_correct = random.choice(correctable_indexes)
        comprehender.language_rep[producer_concept_choice][1][chosen_index_to_correct] = abs(1 - (comprehender.language_rep[producer_concept_choice][1][chosen_index_to_correct]))

    return None
```

1.5 Data-collector functions:

[ ]:
```
# lexical variability
def calculate_pop_lex_var(agent_list, n_concepts):
    pairs_of_agents = itertools.combinations(agent_list, r=2)

    pairs_lex_var = []

    for pair in pairs_of_agents:
        pair_lex_var = calculate_distance(pair, n_concepts)
        pairs_lex_var.append(pair_lex_var)

    pop_av_lex_var = sum(pairs_lex_var) / len(list(itertools.
 ↪combinations(agent_list, r=2)))
    return (pop_av_lex_var)
```

[ ]:
```
def calculate_distance(pair, n_concepts):
    """ per concept per agent pair, distance = 0 if concepts are the same,␣
 ↪distance = 1 if concepts are different
    add up concept distances and divide by total number of concepts """
    concept_lex_var_total = 0  # list of distances between individual concepts␣
 ↪(compare iconic agent a and iconic agent b)
    for n in range(n_concepts):
        if pair[0].language_rep[n][1] != pair[1].language_rep[n][1]:
            concept_lex_var_total += 1  # if concepts don't match, add 1 to␣
 ↪distance

    pair_mean_lex_var = concept_lex_var_total / n_concepts
    return pair_mean_lex_var
```

[ ]:
```
# iconicity
def calculate_degree_of_iconicity(agent):
    concept_iconicity_vals = []

    for concept, meaning_form in agent.language_rep.items():
        comparison_list = ([(p_bit == c_bit) for p_bit, c_bit in␣
 ↪zip(meaning_form[0], meaning_form[1])])  # returns True or False for each␣
 ↪comparison
        concept_iconicity_val = sum(comparison_list) / len(comparison_list)
        concept_iconicity_vals.append(concept_iconicity_val)

    mean_agent_iconicity = sum(concept_iconicity_vals) /␣
 ↪len(concept_iconicity_vals)
    return mean_agent_iconicity
```

[ ]:
```
def calculate_prop_iconicity(agent_list):
    iconicity_list = [a.prop_iconicity for a in agent_list]
```

```
    return sum(iconicity_list) / len(agent_list)
```

1.6 Defining the agent and the model as a whole (using the Mesa package):

[ ]:
```
class ContextAgent(Agent):
    def __init__(self, unique_id, model, n_concepts, n_bits, n_groups):
        super().__init__(unique_id, model)
        self.group = random.choice(range(n_groups))
        self.language_rep = language_skeleton(n_concepts, n_bits)  # dic =␣
↪{concept: [[meaning], [form]}
        self.prop_iconicity = None

    def describe(self):
        #print(f"id = {self.unique_id}, prop iconicity = {self.prop_iconicity},␣
↪group = {self.group}, language = {self.language_rep}")
        print(self.language_rep)

    def step(self):
        self.prop_iconicity = calculate_degree_of_iconicity(self)
```

[ ]:
```
class ContextModel(Model):
    """A model with some number of agents."""
    def __init__(self, n_agents, n_concepts, n_bits, n_groups,␣
↪initial_degree_of_overlap, n_steps, viz_on=False):
        super().__init__()
        self.placement_counter = 0
        self.n_agents = n_agents
        self.n_groups = n_groups
        self.n_concepts = n_concepts
        self.n_bits = n_bits
        self.n_steps = n_steps

        self.current_step = 0
        self.schedule = RandomActivation(self)
        self.running = True  # for server
        self.group_meanings_dic = language_create_meanings(n_concepts, n_bits,␣
↪n_groups) # set up language structure (maybe eventually a class)

        self.width_height = int(sqrt(n_agents))
        self.coordinate_list = list(itertools.product(range(self.width_height),␣
↪range(self.width_height)))  # generate coordinates for grid

        # language game successes and failures
        self.lg_form_success = 0
        self.lg_meaning_success = 0
```

```
        self.lg_bit_update = 0
        self.language_game_stats = {'form_success': None, 'meaning_success':␣
↪None, 'bit_update': None}

        # for datacollector
        self.pop_iconicity = None
        self.pop_lex_var = None
        self.datacollector = DataCollector({'pop_iconicity': 'pop_iconicity',
                                            'pop_lex_var': 'pop_lex_var',
                                            'current_step': 'current_step',
                                            'lg_form_success':␣
↪'lg_form_success',
                                            'lg_meaning_success':␣
↪'lg_meaning_success',
                                            'lg_bit_update': 'lg_bit_update'},
                                           {'group': lambda agent: agent.group,
                                            'language': lambda agent: agent.
↪language_rep,
                                            'prop iconicity': lambda agent:␣
↪agent.prop_iconicity})

        # create agents
        for i in range(self.n_agents):
            a = ContextAgent(i, self, self.n_concepts, self.n_bits, self.
↪n_groups)  # make a new agent
            language_add_meaning(a, self.group_meanings_dic)  # add meaning to␣
↪language skeleton
            language_add_form(a, initial_degree_of_overlap)  # add form to␣
↪language skeleton

            self.schedule.add(a)  # add agent to list of agents
            a.prop_iconicity = calculate_degree_of_iconicity(a)

        self.sorted_agents = sorted(self.schedule.agents, key=lambda agent:␣
↪agent.group)  # sort agents by group

    def collect_data(self):
        self.pop_iconicity = calculate_prop_iconicity(self.schedule.agents)
        self.pop_lex_var = calculate_pop_lex_var(self.schedule.agents, self.
↪n_concepts)
        self.current_step = self.current_step
        self.lg_form_success = self.language_game_stats['form_success']
        self.lg_meaning_success = self.language_game_stats['meaning_success']
        self.lg_bit_update = self.language_game_stats['bit_update']
        self.datacollector.collect(self)
```

```
    def tests(self, a):
        assert len(self.group_meanings_dic) == self.n_groups
        assert len(self.group_meanings_dic[0]) == self.n_concepts
        assert len(self.group_meanings_dic[0][0]) == self.n_bits
        assert len(a.language_rep) == self.n_concepts

    def step(self):
        """ Advance the model by one step """
        self.collect_data()  # set up = year 0

        if self.current_step == 0:
            self.tests(random.choice(self.schedule.agents))  # run tests on a␣
 ↪random agent

        self.current_step += 1
        self.language_game_stats = language_game(self.sorted_agents)  #␣
 ↪language game (only after the set up = year 0)

        #if self.current_step == self.n_steps:
        #    upgma_df = pd.DataFrame()

        #    for i in self.schedule.agents:
        #        for key, value in i.language_rep.items():
        #            new_row = {'id': i.unique_id, 'concept': key, 'form':␣
 ↪value[1]}
        #            upgma_df = upgma_df.append(new_row, ignore_index=True)

        #    upgma_df.to_csv("upgma_data.csv")

        self.schedule.step()
```

[ ]:
```
n_groups = 1

start_time = time.time()

context_model = ContextModel(test_params["n_agents"],␣
 ↪test_params["n_concepts"], test_params["n_bits"],
                             n_groups,␣
 ↪test_params["initial_degree_of_overlap"], test_params["n_steps"])

for i in range(test_params["n_steps"]+1):  # set up = year 0 + x years
    print(i)
    context_model.step()

print("Simulation(s) took %s minutes to run" % round(((time.time() -␣
 ↪start_time) / 60.), 2))  # ADDED BY MW
```

```
df_model_output_1_group = context_model.datacollector.get_model_vars_dataframe()
## alternative option for the agents is get_agent_vars_dataframe(), returns␣
 ↪['Step', 'AgentID', 'neighborhood', 'language', 'prop iconicity']

csv_save_as =␣
 ↪"n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_params["n_bits"])+"_n_agen
df_model_output_1_group = pd.DataFrame(df_model_output_1_group.to_records())  #␣
 ↪gets rid of multiindex
df_model_output_1_group.to_csv(f"{csv_save_as}.csv")
```

[ ]:
```
n_groups = 10

start_time = time.time()

context_model = ContextModel(test_params["n_agents"],␣
 ↪test_params["n_concepts"], test_params["n_bits"],
                              n_groups,␣
 ↪test_params["initial_degree_of_overlap"], test_params["n_steps"])

for i in range(test_params["n_steps"]+1):  # set up = year 0 + x years
    print(i)
    context_model.step()

print("Simulation(s) took %s minutes to run" % round(((time.time() -␣
 ↪start_time) / 60.), 2))  # ADDED BY MW

df_model_output_10_groups = context_model.datacollector.
 ↪get_model_vars_dataframe()
## alternative option for the agents is get_agent_vars_dataframe(), returns␣
 ↪['Step', 'AgentID', 'neighborhood', 'language', 'prop iconicity']

csv_save_as =␣
 ↪"n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_params["n_bits"])+"_n_agen
df_model_output_10_groups = pd.DataFrame(df_model_output_10_groups.
 ↪to_records())  # gets rid of multiindex
df_model_output_10_groups.to_csv(f"{csv_save_as}.csv")
```

[ ]:
```
%matplotlib inline

# colormap
cmap = plt.cm.viridis
cmaplist = [cmap(i) for i in range(cmap.N)]

# set up 2 column figure
fig, (ax0, ax1) = plt.subplots(ncols=2, constrained_layout=True)
```

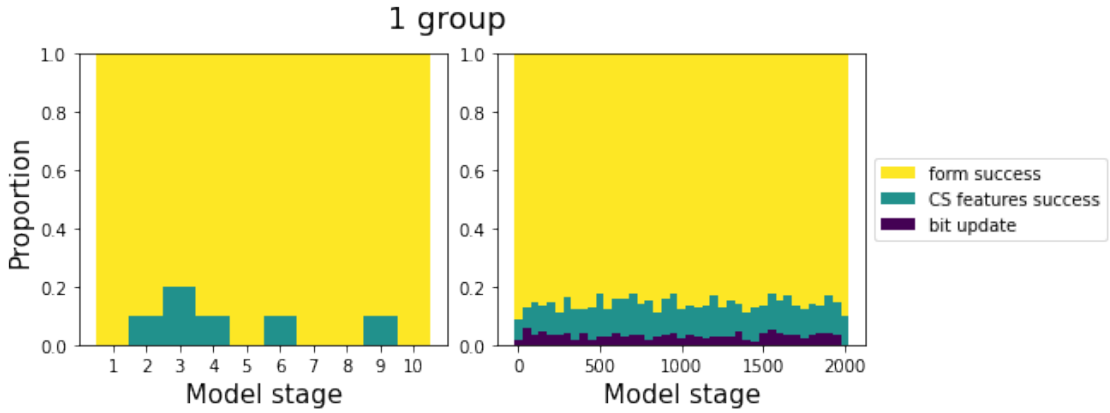
```
fig.set_size_inches(9,3)

# FIG 1 GROUP EXAMPLE RUN
# 1 group, 10 stages on ax0

# Uncomment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_1_group

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(model_output["current_step"].min(), model_output["current_step"].
 ↪max() + 1)  # min, max steps in df
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# axes
ax0.set_xlabel("Model stage", fontsize=15)
ax0.set_ylim(0,1)
ax0.set_ylabel("Proportion", fontsize=15)
ax0.set_xticks(np.arange(1, 11, 1))
```

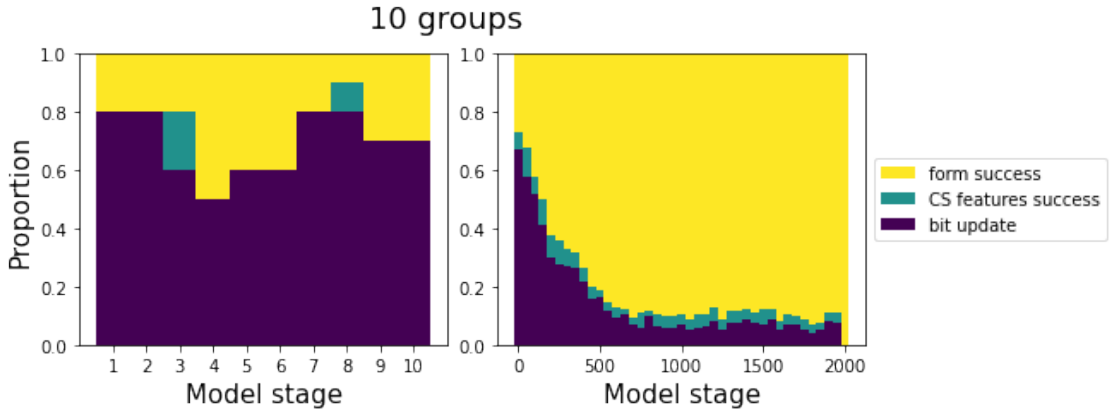
```
# 1 group, 2000 stages on ax1

# Uncomment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_1_group

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# add column with value for groups of 50 (1-50, 51-100, etc.)
for index, row in model_output.iterrows():
    model_output.at[index, "hist_block"] = int(index/50)

model_output_grouped = model_output.groupby(["hist_block"]).mean()
model_output_grouped["original_index"] = model_output_grouped.index * 50
model_output = model_output_grouped[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit", "original_index"]]

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(int(model_output.index.min()), int(model_output.index.max() + 1))␣
 ↪ # min, max steps in df
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
```

```
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# legend
handles, labels = ax1.get_legend_handles_labels()
handles = [handles[2], handles[1], handles[0]]
labels = [labels[2], labels[1], labels[0]]
ax1.legend(handles, labels, loc='center left', bbox_to_anchor=(1, 0.5))

# axes
ax1.set_xlabel("Model stage", fontsize=15)
ax1.set_ylim(0,1)
ax1.set_ylabel("", fontsize=15)
ax1.set_xticks(np.arange(0, 41, step=10))
ax1.set_xticklabels([0,500,1000,1500,2000])

plt.suptitle("1 group", fontsize=18, x=0.4, y=1.1)

plt.savefig("barplot_1group.png", dpi=1000, bbox_inches="tight")



# FIG 10 GROUPS EXAMPLE RUN
# set up 2 column figure
fig, (ax0, ax1) = plt.subplots(ncols=2, constrained_layout=True)
fig.set_size_inches(9,3)

# 10 groups, 10 stages on ax0

# Uncomment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_10_groups

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
```

```
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(model_output["current_step"].min(), model_output["current_step"].
 ↪max() + 1)  # min, max steps in df
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# axes
ax0.set_xlabel("Model stage", fontsize=15)
ax0.set_ylim(0,1)
ax0.set_ylabel("Proportion", fontsize=15)
ax0.set_xticks(np.arange(1, 11, 1))

# 10 groups, 2000 stages on ax1

# Uncomment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_10_groups

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# add column with value for groups of 50 (1-50, 51-100, etc.)
for index, row in model_output.iterrows():
    model_output.at[index, "hist_block"] = int(index/50)
```

```
model_output_grouped = model_output.groupby(["hist_block"]).mean()
model_output_grouped["original_index"] = model_output_grouped.index * 50
model_output = model_output_grouped[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit", "original_index"]]

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(int(model_output.index.min()), int(model_output.index.max() + 1))␣
 ↪ # min, max steps in df
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# legend
handles, labels = ax1.get_legend_handles_labels()
handles = [handles[2], handles[1], handles[0]]
labels = [labels[2], labels[1], labels[0]]
ax1.legend(handles, labels, loc='center left', bbox_to_anchor=(1, 0.5))

# axes
ax1.set_xlabel("Model stage", fontsize=15)
ax1.set_ylim(0,1)
ax1.set_ylabel("", fontsize=15)
ax1.set_xticks(np.arange(0, 41, step=10))
ax1.set_xticklabels([0,500,1000,1500,2000])

plt.suptitle("10 groups", fontsize=18, x=0.4, y=1.1)

plt.savefig("barplot_10groups.png", dpi=1000, bbox_inches="tight")
```





[ ]:
```
n_groups = 5
n_agents = 5

start_time = time.time()

context_model = ContextModel(test_params["n_agents"],␣
 ↪test_params["n_concepts"], test_params["n_bits"],
                              n_groups,␣
 ↪test_params["initial_degree_of_overlap"], test_params["n_steps"])

for i in range(test_params["n_steps"]+1):  # set up = year 0 + x years
    print(i)
    context_model.step()

print("Simulation(s) took %s minutes to run" % round(((time.time() -␣
 ↪start_time) / 60.), 2))  # ADDED BY MW
```

```
df_model_output_5_agents = context_model.datacollector.
 ↪get_model_vars_dataframe()
## alternative option for the agents is get_agent_vars_dataframe(), returns␣
 ↪['Step', 'AgentID', 'neighborhood', 'language', 'prop iconicity']

csv_save_as =␣
 ↪"n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_params["n_bits"])+"_n_agen
df_model_output_5_agents = pd.DataFrame(df_model_output_5_agents.to_records()) ␣
 ↪# gets rid of multiindex
df_model_output_5_agents.to_csv(f"{csv_save_as}.csv")
```

[ ]:
```
#n_groups = 5
n_agents = 100

start_time = time.time()

context_model = ContextModel(test_params["n_agents"],␣
 ↪test_params["n_concepts"], test_params["n_bits"],
                              n_groups,␣
 ↪test_params["initial_degree_of_overlap"], test_params["n_steps"])

for i in range(test_params["n_steps"]+1):  # set up = year 0 + x years
    print(i)
    context_model.step()

print("Simulation(s) took %s minutes to run" % round(((time.time() -␣
 ↪start_time) / 60.), 2))  # ADDED BY MW

df_model_output_100_agents = context_model.datacollector.
 ↪get_model_vars_dataframe()
## alternative option for the agents is get_agent_vars_dataframe(), returns␣
 ↪['Step', 'AgentID', 'neighborhood', 'language', 'prop iconicity']

csv_save_as =␣
 ↪"n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_params["n_bits"])+"_n_agen
df_model_output_100_agents = pd.DataFrame(df_model_output_100_agents.
 ↪to_records())  # gets rid of multiindex
df_model_output_100_agents.to_csv(f"{csv_save_as}.csv")
```

[ ]:
```
%matplotlib inline

# colormap
cmap = plt.cm.viridis
cmaplist = [cmap(i) for i in range(cmap.N)]

# set up 2 column figure
```

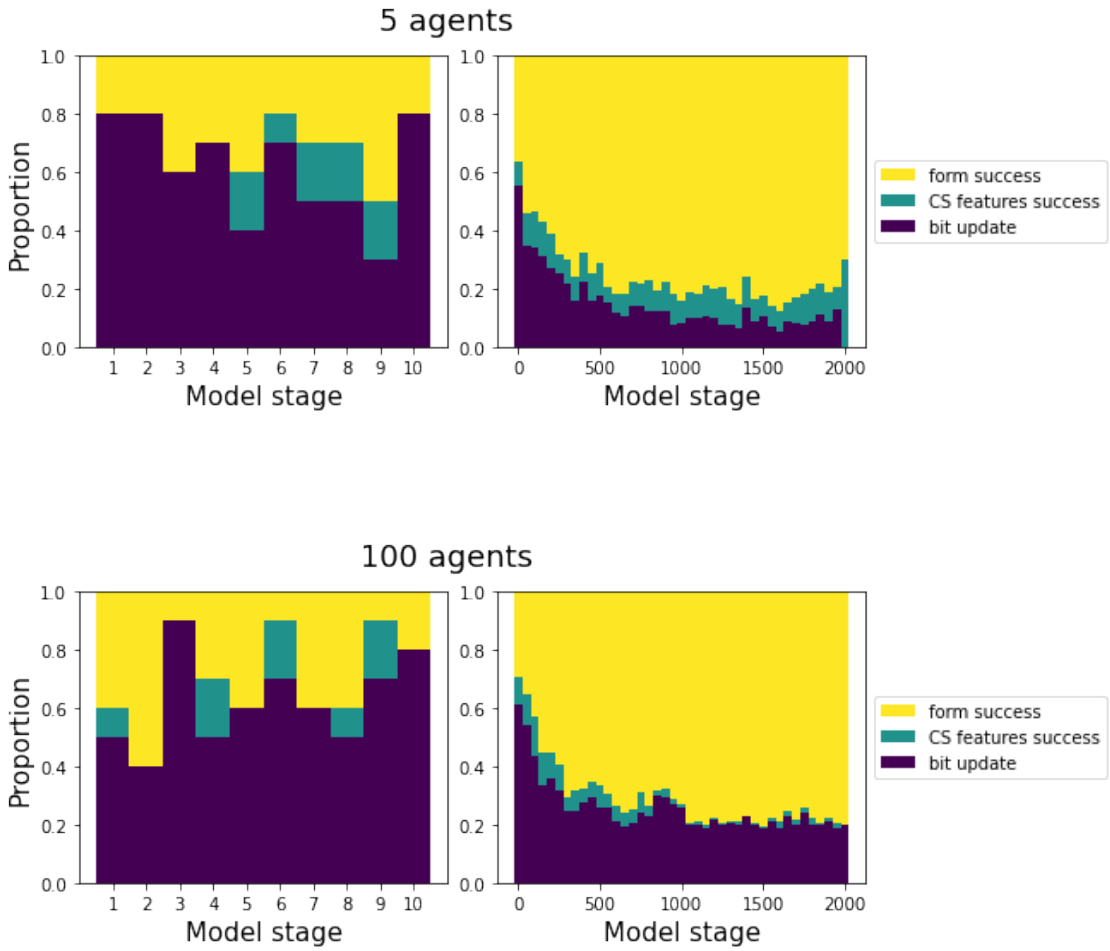
```
fig, (ax0, ax1) = plt.subplots(ncols=2, constrained_layout=True)
fig.set_size_inches(9,3)

# FIG 1 GROUP EXAMPLE RUN
# 5 agents, 10 stages on ax0

# Uncommment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_5_agents

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(model_output["current_step"].min(), model_output["current_step"].
 ↪max() + 1)  # min, max steps in df
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# axes
ax0.set_xlabel("Model stage", fontsize=15)
ax0.set_ylim(0,1)
ax0.set_ylabel("Proportion", fontsize=15)
```

```
ax0.set_xticks(np.arange(1, 11, 1))


# 5 agents, 2000 stages on ax1

# Uncomment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_5_agents

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# add column with value for groups of 50 (1-50, 51-100, etc.)
for index, row in model_output.iterrows():
    model_output.at[index, "hist_block"] = int(index/50)

model_output_grouped = model_output.groupby(["hist_block"]).mean()
model_output_grouped["original_index"] = model_output_grouped.index * 50
model_output = model_output_grouped[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit", "original_index"]]

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(int(model_output.index.min()), int(model_output.index.max() + 1))␣
 ↪ # min, max steps in df
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
```

```
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# legend
handles, labels = ax1.get_legend_handles_labels()
handles = [handles[2], handles[1], handles[0]]
labels = [labels[2], labels[1], labels[0]]
ax1.legend(handles, labels, loc='center left', bbox_to_anchor=(1, 0.5))

# axes
ax1.set_xlabel("Model stage", fontsize=15)
ax1.set_ylim(0,1)
ax1.set_ylabel("", fontsize=15)
ax1.set_xticks(np.arange(0, 41, step=10))
ax1.set_xticklabels([0,500,1000,1500,2000])

plt.suptitle("5 agents", fontsize=18, x=0.4, y=1.1)

plt.savefig("barplot_5agents.png", dpi=1000, bbox_inches="tight")



# FIG 10 GROUPS EXAMPLE RUN
# set up 2 column figure
fig, (ax0, ax1) = plt.subplots(ncols=2, constrained_layout=True)
fig.set_size_inches(9,3)

# 100 agents, 10 stages on ax0

# Uncomment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_100_agents

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
```

```
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(model_output["current_step"].min(), model_output["current_step"].
 ↪max() + 1)  # min, max steps in df
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# axes
ax0.set_xlabel("Model stage", fontsize=15)
ax0.set_ylim(0,1)
ax0.set_ylabel("Proportion", fontsize=15)
ax0.set_xticks(np.arange(1, 11, 1))

# 100 agents, 2000 stages on ax1

# Uncomment the line below if you want to load in your dataframe from a .csv␣
 ↪file:
# model_output = pd.read_csv("", index_col=0)

model_output = df_model_output_100_agents

model_output = model_output[['current_step', 'lg_form_success',␣
 ↪'lg_meaning_success', 'lg_bit_update']]
model_output = model_output.rename(columns={"lg_form_success": "form_success",␣
 ↪"lg_meaning_success": "culturally_salient_features_success", "lg_bit_update":
 ↪ "update_bit"})
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success",␣
 ↪"update_bit"]] = model_output[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit"]].div(10, axis=0)

# add column with value for groups of 50 (1-50, 51-100, etc.)
for index, row in model_output.iterrows():
    model_output.at[index, "hist_block"] = int(index/50)
```

```
model_output_grouped = model_output.groupby(["hist_block"]).mean()
model_output_grouped["original_index"] = model_output_grouped.index * 50
model_output = model_output_grouped[["form_success",␣
 ↪"culturally_salient_features_success", "update_bit", "original_index"]]

# https://www.python-graph-gallery.com/13-percent-stacked-barplot
# From raw value to percentage
totals = [i+j+k for i, j, k in zip(model_output['update_bit'],␣
 ↪model_output['culturally_salient_features_success'],␣
 ↪model_output['form_success'])]
bit_bars = [i / j for i,j in zip(model_output['update_bit'], totals)]
features_bars = [i / j for i,j in␣
 ↪zip(model_output['culturally_salient_features_success'], totals)]
form_bars = [i / j for i,j in zip(model_output['form_success'], totals)]

steps = range(int(model_output.index.min()), int(model_output.index.max() + 1))␣
 ↪ # min, max steps in df
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none",␣
 ↪label="bit update")  # Create green Bars
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1,␣
 ↪edgecolor="none", label="CS features success")  # Create orange Bars
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars,␣
 ↪features_bars)], color=cmaplist[-1], width=1, edgecolor="none", label="form␣
 ↪success")  # Create blue Bars

# legend
handles, labels = ax1.get_legend_handles_labels()
handles = [handles[2], handles[1], handles[0]]
labels = [labels[2], labels[1], labels[0]]
ax1.legend(handles, labels, loc='center left', bbox_to_anchor=(1, 0.5))

# axes
ax1.set_xlabel("Model stage", fontsize=15)
ax1.set_ylim(0,1)
ax1.set_ylabel("", fontsize=15)
ax1.set_xticks(np.arange(0, 41, step=10))
ax1.set_xticklabels([0,500,1000,1500,2000])

plt.suptitle("100 agents", fontsize=18, x=0.4, y=1.1)

plt.savefig("barplot_10agents.png", dpi=1000, bbox_inches="tight")
```



a) As the small population (5 agents) should initially involve a few forms, which in turn lead to fewer bit updates and the retention of iconicity (overlap in the bits between form and meaning), I would expect a relatively higher proportion of CS features success throughout the stages for the small population. In contrast, as the large population (100 agents) initially involves numerous initial forms, the proportion of form updates should be high at the intial stages, and that the proportion of CS features success should decrease as the number of model stages goes up because bit updates lead to the reduction of iconicity.