ABCM Computer lab 4: Variation & Convergence in Populations

In this computer lab, we will explore the agent-based model described in Mudd, K., de Vos, C., & de Boer, B. (2022). Shared Context Facilitates Lexical Variation in Sign Language Emergence. Languages, 7(1), Article 1. https://doi.org/10.3390/languages7010031

Below, we use the Python code written by the first author Katie Mudd herself, which she shared openly on Figshare: https://doi.org/10.6084/m9.figshare.15163872.v1

This code makes use of the Mesa package, which is a Python framework for agent-based modelling. It allows users to quickly create agent-based models using built-in core components (such as spatial grids and agent schedulers) or customized implementations. The code below uses four classes from the Mesa package:

- `Agent`
- `Model`
- `RandomActivation`
- `DataCollector`

The generic `Agent` and `Model` class from the Mesa package are built upon in the code below (using class inheritance) to create the specific type of agent needed for this model (called `ContextAgent` below), and the specific type of model needed (called `ContextModel` below). Using such a child class of the parent class `Model`, the code can then use the built-in `RandomActivation` and `DataCollector` classes from the Mesa package to schedule the agents' interactions and keep track of various measures of the population's interactions and vocabularies.

Let's first install the Mesa package:

In [ ]:
```
pip install mesa
```

Then, let's import all the packages, classes and functions we'll need:

In [ ]:
```
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
```

Parameter settings:

The code cell below contains the parameter settings that we'll need to run simulations further down in the notebook. Below, these parameters are combined in a dictionary, which is the datastructure that the code below expects when retrieving these parameter settings.

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

test_params = dict(
    n_concepts=10, # int: number of concepts
    n_bits=10,  # int: number of bits (determining length of forms and cultural
    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 me
    n_steps=2000  # number of timesteps to run the simulation for (called "mode
)
```

Initialising the population and their language representations

As described on page 7 of Mudd et al. (2022) (see **Initialization** paragraph), each agent in the population has a language representation that consists of `n_concepts` concepts, where each concept is associated with (i) a set of culturally salient features, and (ii) a form. The set of culturally salient features and the form both consist of `n_bits` bits. (Also see Figure 4 in the paper.)

The vector of culturally salient features represents the *meaning* of the concept to the agent (which depends on their cultural background; i.e., the group they belong to). The form vector represents the *form* (i.e., signal) that the agent would use in order to convey that concept. The parameter `initial_degree_of_overlap` determines the degree of overlap between the culturally salient features vector and the form. If this degree of overlap is high, that simulates a form that is highly *iconic* (i.e., when aspects of the form resemble aspects of the meaning).

The code in this section takes care of initialising the population and each agent's language representations, depending on which group in the population they belong to.

This is done using the following four functions:

- `language_skeleton()`: Creates an empty language representation for an agent (where the meaning and form components are initialised with the value `None`)
- `language_create_meanings()`: Randomly generates a bit vector of culturally-salient features for each concept, for each group in the population
- `language_add_meaning()`: Takes a particular agent object, and fills in the meaning vectors in its empty language representation with the bit vectors of culturally-salient features that correspond to the group that the agent belongs to

- `language_add_form()`: Takes a particular agent object, and fills in the form vectors in its empty language representation, depending on the setting of the `initial_degree_of_overlap` parameter, which determines the probability that a given bit of a meaning and form representation are the same (i.e., have the same value at the same index). (See top of page 9 in Mudd et al., 2022.)

In [ ]:
```
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
```

In [ ]:
```
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,
    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_bit
            if len(set(tuple(row) for row in single_group_meaning_list)) == ler
                    single_group_meaning_list):
                condition = True
                group_meaning_dic[n] = single_group_meaning_list

    return group_meaning_dic
```

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

In [ ]:
```
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_ov
            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 (r
            else:
                forms.append(bit)  # append the same bit (iconic)
```

```
        meaning_form[1] = forms
    return agent
```

Running a language game and updating the agents' language representations

As described in pages 9-10 and Figure 5 of Mudd et al. (2022), there are four different stages that a sender-receiver pair can go through in a language game interaction:

1. *signal production*: First, the sender randomly chooses a concept and produces the corresponding form given by the sender's language representation (this form is the signal that the sender produces).
2. *form success*: The receiver finds the form which is closest to the sender's form in the receiver's language representation (by calculating the distance between the sender's form to all forms of the receiver, and choosing the form with the smallest distance). If the concept for that form in the receiver's language representation is the same as the concept that the sender wanted to convey, the pair has achieved *form success* (by using the *conventional link* between a concept and a form). If the pair has achieved *form success*, the language game ends here.
3. *culturally salient features success*: If the pair has *not* achieved *form success*, the receiver now tries to make use of the *iconic–inferential pathway* (where a form and concept are linked via the culturally salient features) in order to interpret the sender's signal (see Figure 2 in Mudd et al., 2022). The receiver does this by comparing the sender's form to all sets of culturally salient features in the receiver's language representation (again by calculating the distance and choosing the concept that has the smallest distance to the sender's form). If the concept chosen in this way is the same as the concept that the sender wanted to convey, the pair has achieved *culturally salient features success*, and the game ends here.
4. *bit update*: If the pair has achieved neither *form success* nor *culturally salient features success*, the receiver proceeds by updating the form that they associate with the concept that the sender wanted to convey. The receiver does this by updating one bit of their form which is different from the form of the sender.

The five functions below take care of the following:

- `language_game()` : Iterates over all agents in the population (sorted by group), assigns them the role of sender, and has them play a language game with a randomly selected other agent from the population. Returns a dictionary that keeps track of how many times the agent pairs in the population reached (i) *form success*, (ii) *culturally salient features success*, or (iii) did a *bit update*
- `language_game_structure()` : Takes a given producer agent and has them play a language game with a randomly selected other agent from the population

- `does_closest_form_match()` : **Corresponds to Step 2 above:** Checks whether the closest form in the receiver's language representation matches the concept that the sender wanted to convey (if so, the agent pair achieves *form success*; see `language_game_structure()`).

- `does_closest_meaning_match()` : **Corresponds to Step 3 above:** Checks whether the meaning vector that most closely matches the sender's form in the receiver's language representation matches the concept that the sender wanted to convey (if so, the agent pair achieves *culturally salient features success*; see `language_game_structure()`).

- `update_comprehender_concept()` : **Corresponds to Step 4 above:** This function is used when the agent pair has achieved neither *form success* nor *culturally salient features success* (see `language_game_structure()`). It takes the form that the receiver associates with the concept that the sender wanted to convey, and changes one bit in that form to make it more similar to the form that the sender associates with that concept.

In [ ]:
```
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": mea
    return language_game_stats
```

In [ ]:
```
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_cho
    if form_match_answer == False:  # 3b
        meaning_match_answer = does_closest_meaning_match(producer, producer_cc
        if meaning_match_answer == False:  # 3b2 (bit update; no success)
            update_comprehender_concept(producer, producer_concept_choice, comp
            return "3b2"
        else:  # 3b1 (cultually salient features success)
            # None
            return "3b1"
    else:  # 3a (form success)
```

```
        # None
        return "3a"
```

In [ ]:
```
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 be
        distance = sum([abs(prod_bit - comp_bit) for prod_bit, comp_bit in zip(
        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_pr
    comp_chosen_form = random.choice(comp_closest_form_list)  # because there c

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

In [ ]:
```
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 be
        distance = sum([abs(prod_bit - comp_bit) for prod_bit, comp_bit in zip(
        distance_from_produced_form[concept] = distance
    comp_closest_meaning = min(distance_from_produced_form, key=distance_from_p

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

In [ ]:
```
def update_comprehender_concept(producer, producer_concept_choice, comprehende
    """ 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 compreher
    comparison_list = ([(p_bit == c_bit) for p_bit, c_bit in zip(producer.langu
    # to prevent case where correct concept has a match for form producer and c
    # this could happen if comprehender has 2 forms which both == form produce
    if all(comparison_list) == True:
        pass
    else:
        correctable_indexes = [i for i, comparison in enumerate(comparison_list
        chosen_index_to_correct = random.choice(correctable_indexes)
        comprehender.language_rep[producer_concept_choice][1][chosen_index_to_c

    return None
```

Exercise 1:

This is a conceptual question: How does the model of what happens in a communicative interaction that is used here differ from the models you've seen in Cuskley et al. (2018) and de Weerd et al. (2015) (i.e., the past two computer labs)?

a) What determines whether a communicative interaction is successful or not, in these three different models?

b) What happens in these three models if a communicative interaction is not immediately successful?

a) In the task in de Weerd et al. (2015), successful communication refers to the situations in which the receiver accurately guess the target position based on producer's moves. In contract, I don't believe there was a clear boudary between success/failure in Cuskley et al. (2018)'s model as they were looking at whether the number of inflections increase or decrease under different population size, turnover, and growth. The present model is rather similar to the one in de Weerd et al. (2015) in the way that there is a clear bondary between success/failure. However, unlike the model in de Weerd et al. (2015) where there is only one way to be successful or not (accurately guess the target location or not), this model allows two ways to succeed: the producer's form corresponds to the receiver's form or the receiver's cultually salient feature.

b) In the first two models, two agents update their representation (i.e, beliefs and vocabulary) immediately after a failure (or interaction in case of the second model). However, in the current model, when the two agents do not succeed in communication (no form success), then they will take another step to establish successful communication (step 3). If this step also fails (no cultually-salient features success), then they will update their representations.

Exercise 2: This is a conceptual question about the *bit update* operation in Mudd et al. (2022) (see Step 4 above).

a) In order for the *bit update* operation to take place, the receiver needs to know what the sender's intended concept was. What process in real-life conversations between people could this map onto? How do we find out what concept someone means, if the word or sign they use for it is different from the one we'd use ourselves?

b) What is it about the reception procedure (i.e. Steps 2 and 3 above) that makes it so that the *bit update* procedure makes this particular sender-receiver pair more likely to reach communicative success the next time they communicate about this same concept, *even* if the receiver's form is still not exactly the same as the sender's form for this concept after *bit update* has taken place?

a) There are various ways for an interlocutor to figure out what concept the other interlocutor means, but one way is to point the referent if it is in the area of physical co-presence. For instance, when the sender means PIG but the receiver doesn't understand due to lack of shared forms or cultually-salient features, the sender can point a pig (deictic gesture) so that the receiver knows what the sender means. Another way is that the sender offers multiple features that are relevant to the concept (e.g., it's an animal that "oinks").

b) As the distance of the sender's form to the receiver's form is calculated by comparing the bits at the same index, and the concept with lowest distance in the receiver's vocabulary will be chosen at step 2, bit updates reduces the distance between sender's and receiver's forms, leading to a higher chance of form success. In addition, if the initial overlap of the sender's forms and cultually salient forms is high, then bit update also leads to a higher chance of cultually salient features success.

Data-collector functions

The functions below are used to keep track of the degree of lexical variability and the degree of iconicity in the population (the measures that are plotted in Figures 9-11 in Mudd et al., 2022). See page 11 (section **Submodel Collect data**) of Mudd et al. (2022) for more details on how these measures are calculated exactly.

In [ ]:
```
# 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
    return (pop_av_lex_var)
```

In [ ]:
```
def calculate_distance(pair, n_concepts):
    """ per concept per agent pair, distance = 0 if concepts are the same, dist
    add up concept distances and divide by total number of concepts """
    concept_lex_var_total = 0  # list of distances between individual concepts
    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 dis

    pair_mean_lex_var = concept_lex_var_total / n_concepts
    return pair_mean_lex_var
```

In [ ]:
```
# 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_fo
        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_
    return mean_agent_iconicity
```

In [ ]:
```
def calculate_prop_iconicity(agent_list):
    iconicity_list = [a.prop_iconicity for a in agent_list]
    return sum(iconicity_list) / len(agent_list)
```

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

The two classes below inherit from the class Agent and Model from the Mesa package.

- The `ContextAgent` class creates an agent, which is an object that consists of the following attributes:
  - unique identifier
  - a group it belongs to
  - a language representation
  - a degree of iconicity of its language representation
- The `ContextModel` class creates a population of `ContextAgent` objects (arranged in groups) and has a method `step()` which steps through 1 timestep of a simulation. A single timestep consists of every agent in the population taking one turn at being a sender in a language game (paired up with a randomly chosen receiver). See also Figure 3 in Mudd et al. (2022).

In [ ]:
```
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 = {cor
        self.prop_iconicity = None

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

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

In [ ]:
```
class ContextModel(Model):
    """A model with some number of agents."""
    def __init__(self, n_agents, n_concepts, n_bits, n_groups, initial_degree_c
        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,

        self.width_height = int(sqrt(n_agents))
        self.coordinate_list = list(itertools.product(range(self.width_height),
```

```
        # 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': No

        # 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_s
                                             'lg_bit_update': 'lg_bit_update'},
                                            {'group': lambda agent: agent.group,
                                             'language': lambda agent: agent.lan
                                             'prop iconicity': lambda agent: age

        # create agents
        for i in range(self.n_agents):
            a = ContextAgent(i, self, self.n_concepts, self.n_bits, self.n_grou
            language_add_meaning(a, self.group_meanings_dic)  # add meaning to
            language_add_form(a, initial_degree_of_overlap)  # add form to lang

            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: age

    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_c
        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 r

        self.current_step += 1
        self.language_game_stats = language_game(self.sorted_agents)  # languag

        #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': valu
        #            upgma_df = upgma_df.append(new_row, ignore_index=True)

        #    upgma_df.to_csv("upgma_data.csv")

        self.schedule.step()
```

Running a single simulation

Now that the `ContextAgent` and `ContextModel` classes have been defined, we can run a single simulation run. The number of timesteps for which a simulation runs is determined by the `n_steps` parameter (defined at the top of this notebook as one of the key-value pairs in the `test_params` dictionary).

The code below shows how you can run a single simulation for a population that consists of only 1 group. The resulting dataframe is saved in the variable `df_model_output_1_group`, but also in a .csv file in your current working directory. If you want to open the dataframe again later from the .csv file, use the read_csv() method of the Pandas dataframe as follows:

`my_dataframe = pandas.read_csv("my_filename.csv")`

In [ ]:
```
n_groups = 1

start_time = time.time()

context_model = ContextModel(test_params["n_agents"], test_params["n_concepts"]
                             n_groups, test_params["initial_degree_of_overlap"]

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

df_model_output_1_group = context_model.datacollector.get_model_vars_dataframe(
## alternative option for the agents is get_agent_vars_dataframe(), returns ['S

csv_save_as = "n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_
df_model_output_1_group = pd.DataFrame(df_model_output_1_group.to_records())  #
df_model_output_1_group.to_csv(f"{csv_save_as}.csv")
```

Let's first inspect the resulting dataframe:

In [ ]:
```
df_model_output_1_group
```

Out[ ]:

	index	pop_iconicity	pop_lex_var	current_step	lg_form_success	lg_meaning_success
0	0	0.951	0.664444	0	NaN	NaN
1	1	0.951	0.664444	1	8.0	2.0
2	2	0.951	0.664444	2	9.0	1.0
3	3	0.951	0.664444	3	10.0	0.0
4	4	0.952	0.651111	4	9.0	0.0
...	...	...	...	...	...	...
1996	1996	0.938	0.586667	1996	8.0	2.0
1997	1997	0.938	0.586667	1997	10.0	0.0
1998	1998	0.938	0.586667	1998	10.0	0.0
1999	1999	0.938	0.586667	1999	9.0	1.0
2000	2000	0.938	0.586667	2000	9.0	1.0

2001 rows × 7 columns

Now, we can run a similar simulation, but for a population that consists of 10 groups (where each group has different culturally salient features for each concept):

In [ ]:

```
n_groups = 10

start_time = time.time()

context_model = ContextModel(test_params["n_agents"], test_params["n_concepts"]
                             n_groups, test_params["initial_degree_of_overlap"]

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

df_model_output_10_groups = context_model.datacollector.get_model_vars_datafram
## alternative option for the agents is get_agent_vars_dataframe(), returns ['S

csv_save_as = "n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_
df_model_output_10_groups = pd.DataFrame(df_model_output_10_groups.to_records()
df_model_output_10_groups.to_csv(f"{csv_save_as}.csv")
```

Let's again inspect the resulting dataframe:

In [ ]:

```
df_model_output_10_groups
```

Out[ ]:

	index	pop_iconicity	pop_lex_var	current_step	lg_form_success	lg_meaning_success
0	0	0.948	0.940000	0	NaN	NaN
1	1	0.945	0.942222	1	7.0	0.0
2	2	0.941	0.946667	2	6.0	0.0
3	3	0.936	0.948889	3	5.0	0.0
4	4	0.929	0.957778	4	3.0	0.0
...	...	...	...	...	...	...
1996	1996	0.589	0.266667	1996	8.0	0.0
1997	1997	0.589	0.266667	1997	8.0	0.0
1998	1998	0.589	0.266667	1998	10.0	0.0
1999	1999	0.589	0.266667	1999	9.0	0.0
2000	2000	0.589	0.266667	2000	9.0	1.0

2001 rows × 7 columns

Plotting what happens inside language games (1 group vs. 10 groups)

The code below creates plots like the ones in Figures 7 and 8 in Mudd et al. (2022), showing what happened in the language games in the two simulations we ran above. Remember that these two simulations differ only in one parameter: the number of groups (`n_groups`), which determines which set of culturally salient features an agent has. The plots that are created below show the proportion of language game outcomes (i.e., how often pairs of agents achieved *form success*, *culturally salient features success* or ended up doing a *bit update*). The resulting plots are shown below the code cell, but also saved as .png files to your current working directory (see lines using `plt.savefig()` method below).

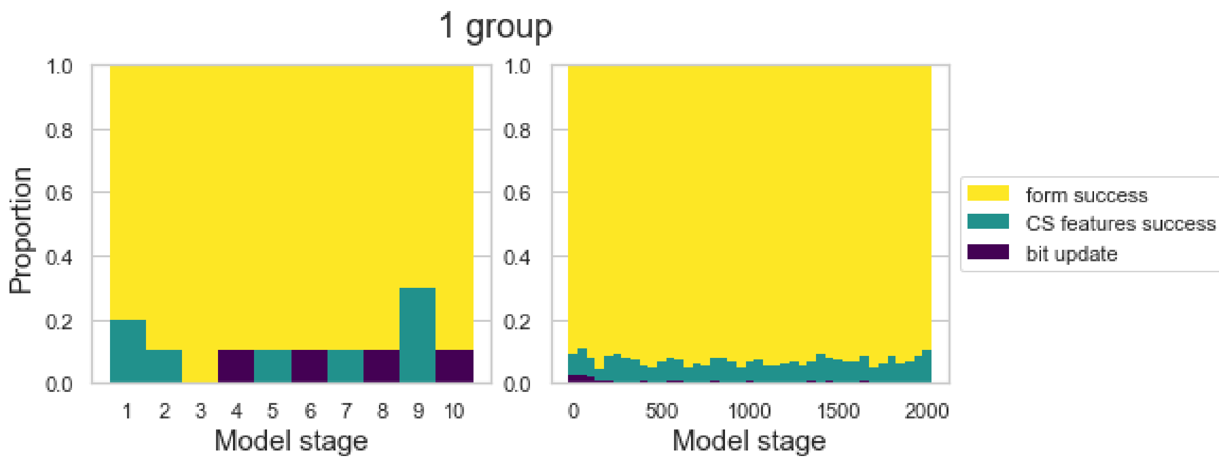
In [ ]:

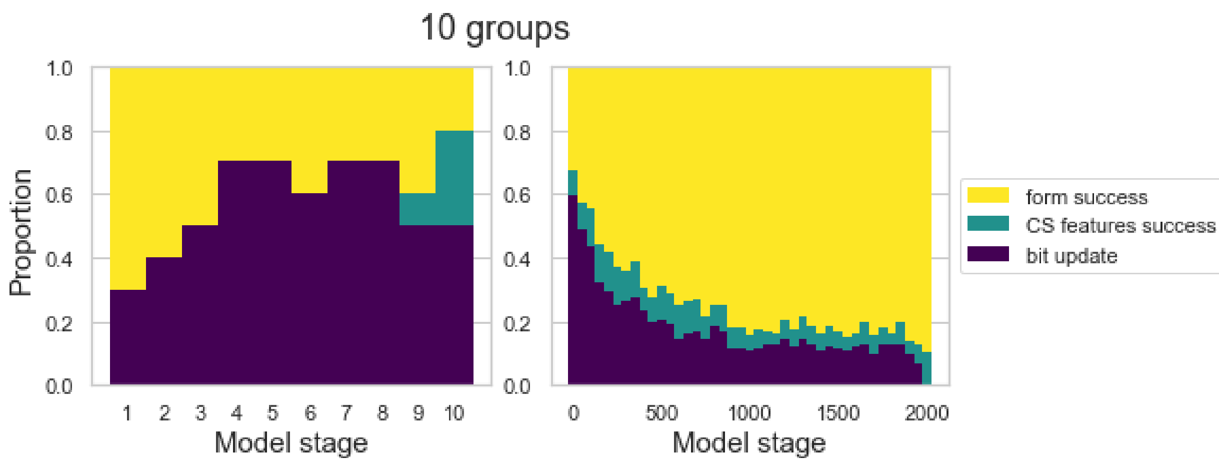
```
%matplotlib inline

# 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 fi
# 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_suc
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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['cu
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
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"].
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ec
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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 fi
# 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_suc
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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_featur

# 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['cu
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
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))
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ec
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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 fi
# 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_suc
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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['cu
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
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"].
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ec
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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 fi
# 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_suc
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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_featu

# 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['cu
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
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))
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ed
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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")
```





Exercise 3:

Run two simulations, both with `n_groups = 10`, in which instead of varying the number of groups in the population, you instead vary the `initial_degree_of_overlap` parameter. Try out what happens with the following settings:

- `initial_degree_of_overlap` = 0.9
- `initial_degree_of_overlap` = 0.5

Plot what happens in the language games, just like we did above. Explain what you see in the plots comparing the two different parameter settings, and try to explain why that is.

I've copy-pasted the code cell with parameter settings from the top of the notebook below, so that you can easily change the parameters as needed:

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

test_params = dict(
    n_concepts=10, # int: number of concepts
    n_bits=10,  # int: number of bits (determining length of forms and cultural
    n_agents=10, # int: number of agents in the population
    n_groups=10,  # determines how many different semantic groups there are
    n_steps=2000,  # number of timesteps to run the simulation for (called "mod
    initial_degree_of_overlap=0.9  # degree of overlap between the form and mea
)
```

In [ ]:
```
start_time = time.time()

context_model = ContextModel(test_params["n_agents"], test_params["n_concepts"]
                             n_groups, test_params["initial_degree_of_overlap"]

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

df_model_output_09_overlap = context_model.datacollector.get_model_vars_datafra
## alternative option for the agents is get_agent_vars_dataframe(), returns ['S

csv_save_as = "n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_
df_model_output_09_overlap = pd.DataFrame(df_model_output_09_overlap.to_records
df_model_output_09_overlap.to_csv(f"{csv_save_as}.csv")
```

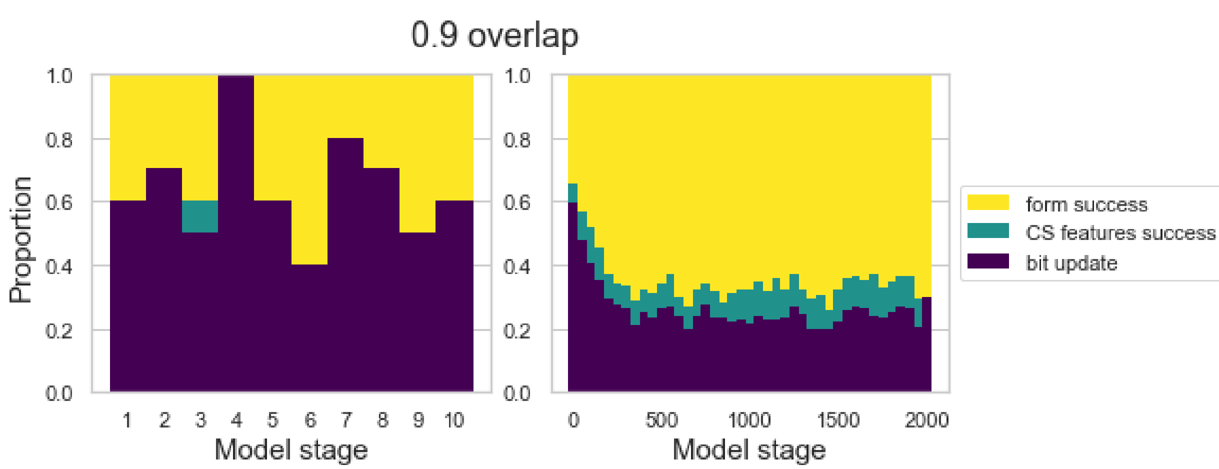
In [ ]:
```
test_params = dict(
    n_concepts=10, # int: number of concepts
    n_bits=10,  # int: number of bits (determining length of forms and cultural
    n_agents=10, # int: number of agents in the population
    n_groups=10,  # determines how many different semantic groups there are
    n_steps=2000,  # number of timesteps to run the simulation for (called "mod
    initial_degree_of_overlap=0.3  # degree of overlap between the form and mea
)

start_time = time.time()

context_model = ContextModel(test_params["n_agents"], test_params["n_concepts"]
                             n_groups, test_params["initial_degree_of_overlap"]

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

df_model_output_05_overlap = context_model.datacollector.get_model_vars_datafra
## alternative option for the agents is get_agent_vars_dataframe(), returns ['S

csv_save_as = "n_concepts_"+str(test_params["n_concepts"])+"_n_bits_"+str(test_
df_model_output_05_overlap = pd.DataFrame(df_model_output_05_overlap.to_records
df_model_output_05_overlap.to_csv(f"{csv_save_as}.csv")
```

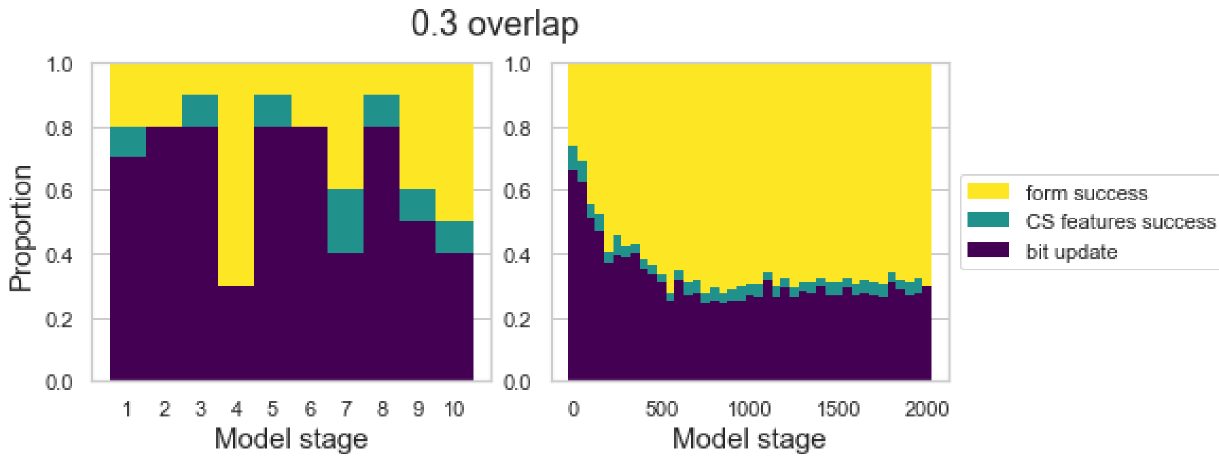
In [ ]:
```
%matplotlib inline

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

#initial_degree_of_overlap = 0.9

model_output = df_model_output_09_overlap

model_output = model_output[['current_step', 'lg_form_success', 'lg_meaning_suc
```

```
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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['cu
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
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"].
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ed
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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))


# 0.9 overlap, 2000 stages on ax1

model_output = df_model_output_09_overlap

model_output = model_output[['current_step', 'lg_form_success', 'lg_meaning_suc
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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_featur

# 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['cu
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
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))
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ed
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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("0.9 overlap", fontsize=18, x=0.4, y=1.1)

plt.savefig("barplot_09overlap.png", dpi=1000, bbox_inches="tight")



################################################################################
# 0.5 overlap, 10 stages on ax0

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

model_output = df_model_output_05_overlap

model_output = model_output[['current_step', 'lg_form_success', 'lg_meaning_suc
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.iloc[1:11]
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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['cu
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
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"].
ax0.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax0.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ed
ax0.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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))

# 0.3 overlap, 2000 stages on ax1

model_output = df_model_output_05_overlap

model_output = model_output[['current_step', 'lg_form_success', 'lg_meaning_suc
model_output = model_output.rename(columns={"lg_form_success": "form_success",
model_output = model_output.drop([0])
model_output[["form_success", "culturally_salient_features_success", "update_bi

# 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_featu

# 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['cu
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
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))
ax1.bar(steps, bit_bars, color=cmaplist[0], width=1, edgecolor="none", label="b
ax1.bar(steps, features_bars, bottom=bit_bars, color=cmaplist[128], width=1, ed
ax1.bar(steps, form_bars, bottom=[i + j for i, j in zip(bit_bars, features_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("0.3 overlap", fontsize=18, x=0.4, y=1.1)

plt.savefig("barplot_05overlap.png", dpi=1000, bbox_inches="tight")
```





The plots show that a lower intial_degree_of_overlap is associated with less CS features success. This makes sense because the agents are less likely to end in cultually salient features success as the initial_degree_of_overlap decreases.

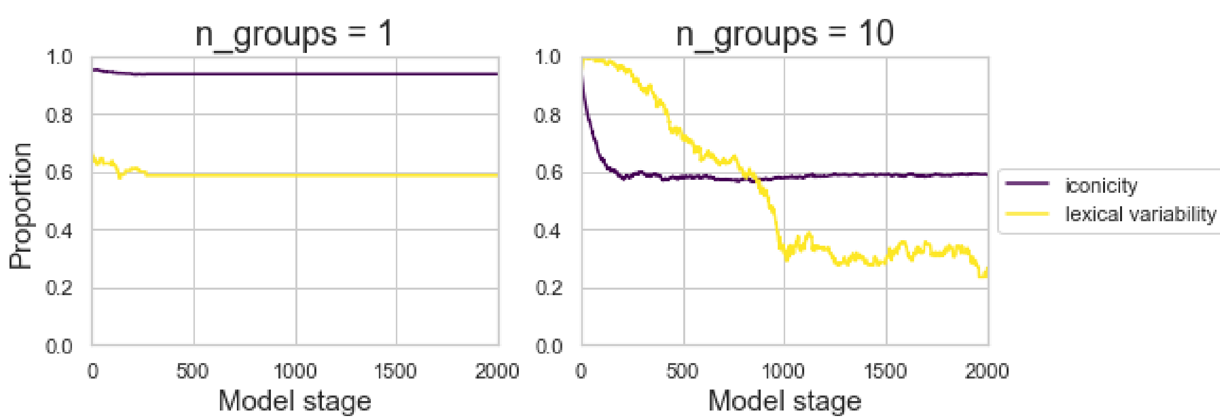
Plotting results of single simulation runs (1 group vs. 10 groups)

The code cell below takes the simulation results of the simulations with 1 and 10 groups we ran above (both with `initial_degree_of_overlap=0.9`) and plots how the degrees of lexical variability and iconicity change over time in the population, similar to Figure 9 in Mudd et al. (2022). The resulting plot is again shown as output below the code cell, but also saved as a .png file.

In [ ]:

```
%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)

# # n_groups = 1
# model_output = pd.read_csv("", index_col=0)  # pop_iconicity, pop_lex_var, ye

model_output = df_model_output_1_group

model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step"]]
model_output = pd.melt(model_output, id_vars=["current_step"])
model_output = model_output.rename(columns={"variable": "measure"})  # year, ru
model_output["measure"] = model_output["measure"].map({"pop_iconicity": "iconic

sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="value", hue="measure", ci=

# axes
ax0.set_title("n_groups = 1", fontsize=18)
```

```
ax0.set_xlim(0,2000)
ax0.set_xlabel("Model stage", fontsize=15)
ax0.set_ylim(0,1)
ax0.set_ylabel("Proportion", fontsize=15)
ax0.get_legend().remove()


# # n_groups = 10
# model_output = pd.read_csv("", index_col=0)  # pop_iconicity, pop_lex_var, ye

model_output = df_model_output_10_groups

model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step"]]
model_output = pd.melt(model_output, id_vars=["current_step"])
model_output = model_output.rename(columns={"variable": "measure"})  # year, ru
model_output["measure"] = model_output["measure"].map({"pop_iconicity": "iconic

sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="value", hue="measure", ci=
ax1.set(xlabel="Model stage", ylabel="Proportion", ylim=(0,1), xlim=(0, 2000))
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))  # Add a legend

# axes
ax1.set_title("n_groups = 10", fontsize=18)
ax1.set_xlim(0,2000)
ax1.set_xlabel("Model stage", fontsize=15)
ax1.set_ylim(0,1)
ax1.set_ylabel("", fontsize=15)

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

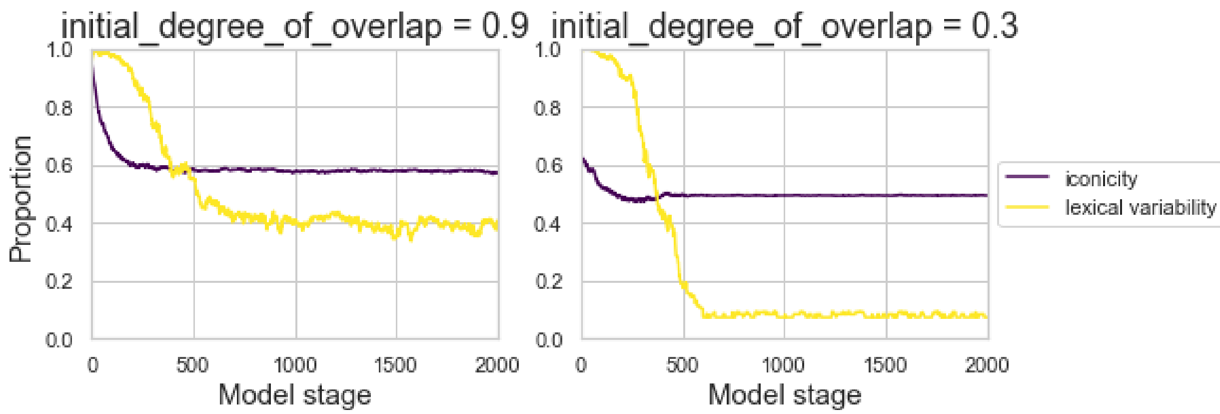


Exercise 4:

Take the simulations you ran for exercise 3, with the two different settings of the `initial_degree_of_overlap` parameter, and use the plotting code above in order to plot how the degree of lexical variability and iconicity change over time in the two simulations. Describe the differences you see between the two plots, and try to explain them.

In [ ]:
```
%matplotlib inline

# set up 2 column figure
```

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

# # initial_degree_of_overlap = 0.9
# model_output = pd.read_csv("", index_col=0)  # pop_iconicity, pop_lex_var, ye

model_output = df_model_output_09_overlap

model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step"]]
model_output = pd.melt(model_output, id_vars=["current_step"])
model_output = model_output.rename(columns={"variable": "measure"})  # year, ru
model_output["measure"] = model_output["measure"].map({"pop_iconicity": "iconic

sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="value", hue="measure", ci=

# axes
ax0.set_title("initial_degree_of_overlap = 0.9", fontsize=18)
ax0.set_xlim(0,2000)
ax0.set_xlabel("Model stage", fontsize=15)
ax0.set_ylim(0,1)
ax0.set_ylabel("Proportion", fontsize=15)
ax0.get_legend().remove()


# # initial_degree_of_overlap = 0.3
# model_output = pd.read_csv("", index_col=0)  # pop_iconicity, pop_lex_var, ye

model_output = df_model_output_05_overlap

model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step"]]
model_output = pd.melt(model_output, id_vars=["current_step"])
model_output = model_output.rename(columns={"variable": "measure"})  # year, ru
model_output["measure"] = model_output["measure"].map({"pop_iconicity": "iconic

sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="value", hue="measure", ci=
ax1.set(xlabel="Model stage", ylabel="Proportion", ylim=(0,1), xlim=(0, 2000))
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))  # Add a legend

# axes
ax1.set_title("initial_degree_of_overlap = 0.3", fontsize=18)
ax1.set_xlim(0,2000)
ax1.set_xlabel("Model stage", fontsize=15)
ax1.set_ylim(0,1)
ax1.set_ylabel("", fontsize=15)

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



The figures show that the lower the initial_degree_of_overlap is, the lower the lexical variability is in the later stages. This could be because less iconicity leads to less chance to end in CS features success and to more frequent bit updates, which result in uniformity in lemma forms.

Running a batch of simulations

Below we first need to set some extra parameters in order to:

- Run several conditions in which we vary the number of groups in the population (while keeping all other parameter settings constant)
- Run a number of independent simulation runs per condition (i.e., per setting of the `n_groups` parameter)

In [ ]:

```
################# MORE PARAMETER SETTINGS: #################

# The code below turns n_groups into a variable parameter; to run several diffe
variable_params = dict(n_groups=[1, 2, 5, 10]) # the different numbers of group
# need to do this because otherwise fixed_parameters overwrites variable_parame
fixed_params = dict((k, v) for (k, v) in test_params.items() if k not in variab

n_iterations = 20  # number of independent simulation runs per condition (calle
```

In order to run a batch run with 4 different group sizes in a reasonable amount of time, we have to lower the `n_iterations` parameter (which determines how many independent simulation runs are run per condition) compared to the original paper. Mudd et al. (2022) used 100 runs per condition. This gives them a solid idea of how much variation there is between independent simulation runs (as some parts of the simulations are probabilistic/stochastic in nature). In the code cell above, I set `n_iterations` to 20. With this setting, the batch run below took about 5 minutes to run on my Macbook Pro which has a 2,6 GHz 6-Core Intel Core i7 processor. If the batch run still hasn't finished running after 10 minutes on your computer, consider decreasing the `n_iterations` parameter further; for example to 10.

Or, if you want to get a better idea of the variability between runs, and you have some time to wait for the simulations to finish running, you can increase the `n_iterations` parameter.

In [ ]:
```
def create_batch_runner(model, variable_parameters=None, **kwargs):
    """ function created to circumvent problem of not having any variable_paran
    even though mesa documentation says default of variable_parameters=None
    there is an error if None is passed... Yannick wrote mesa to fix this"""
    if not variable_parameters:
        return FixedBatchRunner(model, parameters_list=[], **kwargs)
    else:
        return BatchRunner(model, variable_parameters=variable_parameters, **kv
```

In [ ]:
```
br = create_batch_runner(ContextModel,
                         variable_parameters=variable_params,
                         fixed_parameters=fixed_params,
                         iterations=n_iterations,
                         max_steps=test_params["n_steps"]+1,  # set up = year (
                         model_reporters={"Data Collector": lambda m: m.datacol
```

```
/var/folders/yr/vf28v1mn2y1ck9bp7ljl0p8h0000gn/T/ipykernel_44962/3310961209.p
y:8: DeprecationWarning: BatchRunner class has been replaced by batch_run func
tion. Please see documentation.
  return BatchRunner(model, variable_parameters=variable_parameters, **kwargs)
```

In [ ]:
```
start_time = time.time()

br.run_all()
br_df = br.get_model_vars_dataframe()  # df with params + data collector per ru
br_step_data = pd.DataFrame()

for idx, row in br_df.iterrows():
    assert isinstance(row["Data Collector"], DataCollector)
    i_run_data = row["Data Collector"].get_model_vars_dataframe()
    i_run_data['idx'] = idx
    br_step_data = br_step_data.append(i_run_data, ignore_index=True)

final_df = br_step_data.join(br_df.drop("Data Collector", axis="columns"), on='
final_df = final_df.rename(columns={"idx": "run"})
final_df.to_csv(f"{csv_save_as}.csv")

print("Simulation(s) took %s minutes to run" % round(((time.time() - start_time
```

Let's first inspect the resulting dataframe:

In [ ]:
```
final_df
```

Out[ ]:

	pop_iconicity	pop_lex_var	current_step	lg_form_success	lg_meaning_success	lg_b
0	0.663	0.991111	0	NaN	NaN	
1	0.666	0.993333	1	1.0	2.0	
2	0.662	0.995556	2	5.0	1.0	
3	0.665	0.993333	3	2.0	1.0	
4	0.665	0.993333	4	2.0	2.0	
...	...	...	...	...	...	
160075	0.457	0.235556	1996	10.0	0.0	
160076	0.457	0.235556	1997	9.0	0.0	
160077	0.457	0.235556	1998	9.0	0.0	
160078	0.457	0.235556	1999	10.0	0.0	
160079	0.457	0.235556	2000	10.0	0.0	

160080 rows × 14 columns

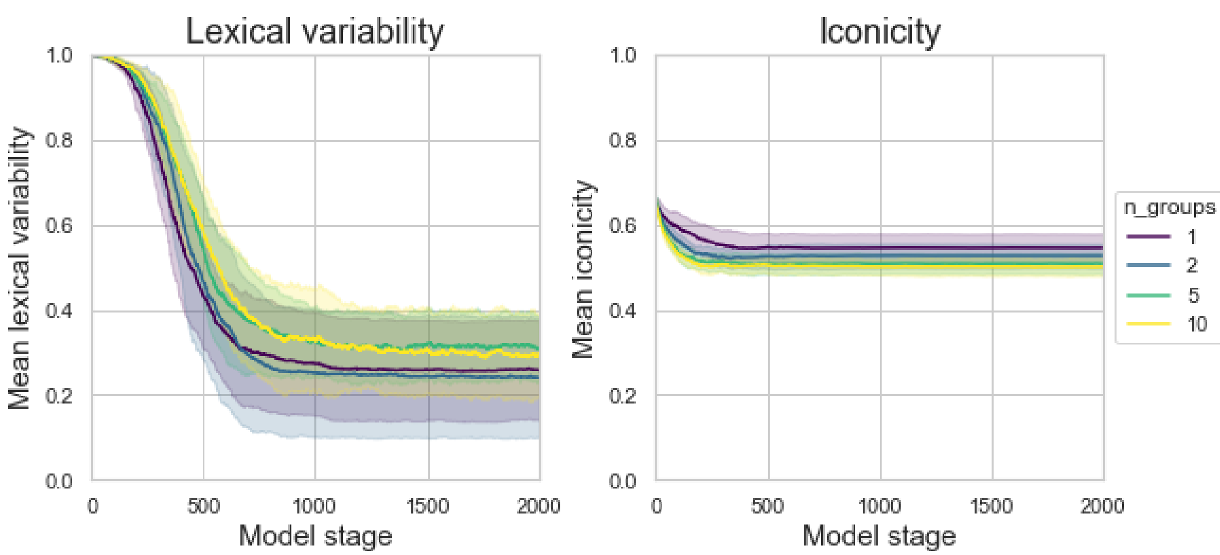
Plotting the results of a batch of simulations

The code below plots the degrees of lexical variability and iconicity over time for each parameter setting included in your batch run. Given that we have now run 20 independent simulation runs per condition, the plots below show both the mean (dark line) and standard deviations (shaded areas) over those 20 independent runs. These plots are the same as Figure 10 in Mudd et al. (2022).

In [ ]:

```
%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, 4)

# N_GROUPS
# model_output = pd.read_csv("", index_col=0)  # pop_iconicity, pop_lex_var, ye
model_output = final_df

model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step", "r

# lexical variability
sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="pop_lex_var", hue="n_group
# axes
ax0.set_title("Lexical variability", fontsize=18)
ax0.set_xlim(0,2000)
ax0.set_xlabel("Model stage", fontsize=15)
```

```
ax0.set_ylim(0,1)
ax0.set_ylabel("Mean lexical variability", fontsize=15)
ax0.get_legend().remove()

# iconicity
sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="pop_iconicity", hue="n_grc
# axes
ax1.set_title("Iconicity", fontsize=18)
ax1.set_xlim(0,2000)
ax1.set_xlabel("Model stage", fontsize=15)
ax1.set_ylim(0,1)
ax1.set_ylabel("Mean iconicity", fontsize=15)
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5), title="n_groups")  # Adc

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



Exercise 5:

Perform a batch run like the one above, but instead of varying the `n_groups` parameter, vary the `initial_degree_of_overlap` parameter instead, using the following values:

- `initial_degree_of_overlap = 0.1`
- `initial_degree_of_overlap = 0.3`
- `initial_degree_of_overlap = 0.6`
- `initial_degree_of_overlap = 0.9`

Throughout each of these simulations, fix the value of the `n_groups` parameter at 10.

Plot the results of your batch run using the plotting code above (which plots the degree of lexical variability and the degree of iconicity for each of the four different parameter settings together). This requires setting the `hue` input argument of the sns.lineplot() function to `hue="initial_degree_of_overlap"` instead of `hue="n_groups"`. It also requires changing the line:

`model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step", "run", "n_groups"]]`

to:

`model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step", "run", "initial_degree_of_overlap"]]`

a) Describe the differences that you see as a result of the different settings and `initial_degree_of_overlap` parameter, and try to explain them.

b) (Conceptual question:) In what ways is manipulating the `initial_degree_of_overlap` parameter in this model different from manipulating the `n_groups` parameter? And in what ways might they be getting at the same thing?

To help you along with the first step, I've copy-pasted the two code cells with parameter settings below. The first code cell allows you to change the fixed parameters, and the second code cells allows you to change the variable parameters, which should be varied in the batch run. In the simulation above, the `variable_params` dictionary was used to vary the `n_groups` parameter, but for Exercise 5, you want to vary the `initial_degree_of_overlap` parameter instead.

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

test_params = dict(
    n_concepts=10, # int: number of concepts
    n_bits=10,  # int: number of bits (determining length of forms and cultural
    n_agents=10, # int: number of agents in the population
    n_groups=10,  # determines how many different semantic groups there are
    initial_degree_of_overlap=0.1,  # degree of overlap between the form and me
    n_steps=2000  # number of timesteps to run the simulation for (called "mode
)
```

In [ ]:
```
################# MORE PARAMETER SETTINGS: #################

# The code below turns n_groups into a variable parameter; to run several diffe
variable_params = dict(initial_degree_of_overlap=[0.1, 0.3, 0.6, 0.9]) # the di
# need to do this because otherwise fixed_parameters overwrites variable_parame
fixed_params = dict((k, v) for (k, v) in test_params.items() if k not in variab

n_iterations = 20  # number of independent simulation runs per condition (calle
```

In [ ]:
```
br = create_batch_runner(ContextModel,
                         variable_parameters=variable_params,
                         fixed_parameters=fixed_params,
                         iterations=n_iterations,
                         max_steps=test_params["n_steps"]+1,  # set up = year 0
                         model_reporters={"Data Collector": lambda m: m.datacol
```

```
/var/folders/yr/vf28v1mn2y1ck9bp7ljl0p8h0000gn/T/ipykernel_44962/3310961209.p
y:8: DeprecationWarning: BatchRunner class has been replaced by batch_run func
tion. Please see documentation.
  return BatchRunner(model, variable_parameters=variable_parameters, **kwargs)
```

In [ ]:

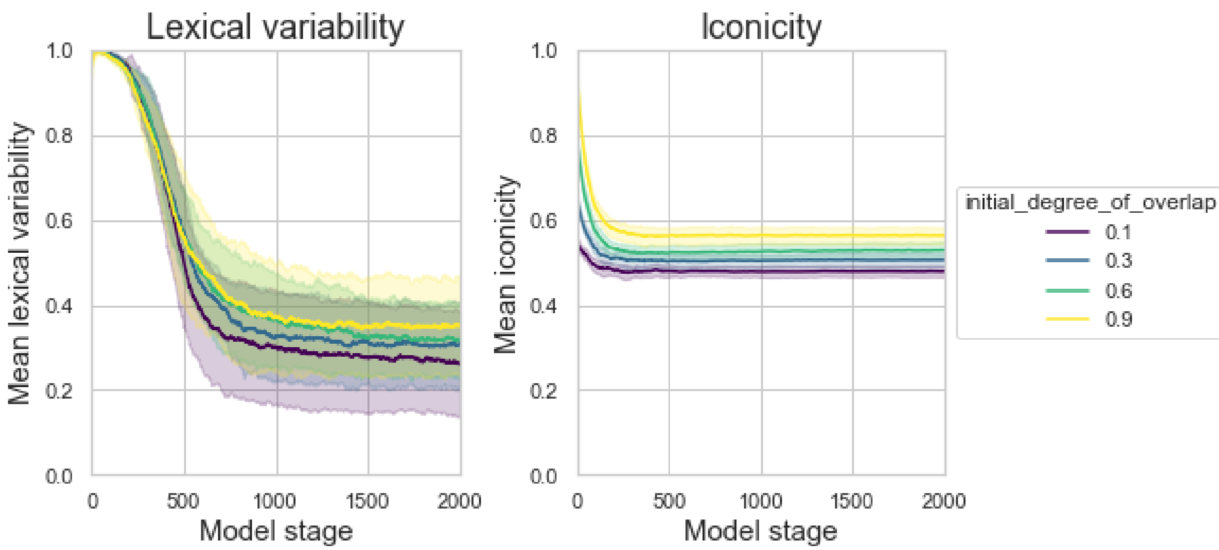
```
start_time = time.time()

br.run_all()
br_df = br.get_model_vars_dataframe()  # df with params + data collector per ru
br_step_data = pd.DataFrame()

for idx, row in br_df.iterrows():
    assert isinstance(row["Data Collector"], DataCollector)
    i_run_data = row["Data Collector"].get_model_vars_dataframe()
    i_run_data['idx'] = idx
    br_step_data = br_step_data.append(i_run_data, ignore_index=True)

final_df = br_step_data.join(br_df.drop("Data Collector", axis="columns"), on='
final_df = final_df.rename(columns={"idx": "run"})
final_df.to_csv(f"{csv_save_as}.csv")

print("Simulation(s) took %s minutes to run" % round(((time.time() - start_time
```

In [ ]:

```
%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, 4)

# N_GROUPS
# model_output = pd.read_csv("", index_col=0)  # pop_iconicity, pop_lex_var, ye
model_output = final_df

model_output = model_output[["pop_iconicity", "pop_lex_var", "current_step", "r

# lexical variability
sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="pop_lex_var", hue="initial
# axes
ax0.set_title("Lexical variability", fontsize=18)
ax0.set_xlim(0,2000)
ax0.set_xlabel("Model stage", fontsize=15)
ax0.set_ylim(0,1)
ax0.set_ylabel("Mean lexical variability", fontsize=15)
ax0.get_legend().remove()

# iconicity
sns.set(style='whitegrid')
sns.lineplot(data=model_output, x="current_step", y="pop_iconicity", hue="initi
# axes
ax1.set_title("Iconicity", fontsize=18)
ax1.set_xlim(0,2000)
ax1.set_xlabel("Model stage", fontsize=15)
```

```
ax1.set_ylim(0,1)
ax1.set_ylabel("Mean iconicity", fontsize=15)
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5), title="initial_degree_of

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

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



a) Both lexical variability and iconicity decreases as the inititial_degree_of_overlap decreases, although the degree of to which lexical variability decreases is much more prominant than iconicity. This might be because less iconicity leads to fewer chance to end in CS features success and to more frequent bit updates, which in turn result in less variability in lemma forms. The reason why iconicity doesn't drop below 0.5 is because it is calculated bit-by-bit, and thus 0.5 means chance level.

b) Although manipulating the initial_degree_of_overlap and n_groups parameter both affect lexical variability and iconicity in a positively correlated way (higher initial_degree_of_overlap and n_groups is associated with higher lexical variability and iconicity), they are fundamentally different. One major difference is that decreasing/increasing initial_degree_of_overlap result in lower/higher iconicity from the very beginning, whareas decreasing/increasing n_groups does not affect iconicity in the beginning, as members of each group has rather iconic mapping between concept and form (assuming that initial_degree_of_overlap is high) before interacting with people from other groups. As such, in the case of initial_degree_of_overlap manipulation, lexical varibility decreases because the agents within/across groups don't have shared forms due to less iconic mapping between meaning and forms, while in the case of n_groups manipulation, lexical varibility decreases because the agents across groups but not within each group don't have shared forms because each group has their own CS features that are shared by other groups.