ABCM Computer lab 3: Population effects & Cultural evolution

This notebook contains a reproduction of the model by Cuskley et al. (2018) in python. Below follows a brief walk-through of the code.

To load the code into your notebook, make sure to run each of the code cells below in turn.

First, let's import the necessary packages:

In [ ]:
```
import random
import numpy as np
import pandas as pd
import time
import matplotlib.pyplot as plt
import seaborn as sns
```

We start by setting a bunch of parameters. Unfortunately, the code takes a while to run. To make it feasible to run some simulations in a reasonable amount of time, the code below therefore makes a number of changes compared to the Cuskley et al. (2018) parameter settings. See the parameter settings below; the comment after each parameter states what setting Cuskley et al. (2018) used.

These measures should hopefully allow you to run the relevant simulations in around 15 min per condition.

Have a look at each of the parameters below, and check whether you understand which parameter or condition described in Cuskley et al. (2018) they correspond to.

In [ ]:
```
n_runs = 2  # int: number of independent simulation runs. Cuskley et al. (2018)
pop_sizes = [20, 100]  # list of ints: initial pop sizes. Cuskley et al. (2018)
n_lemmas = 14  # int: number of lemmas. Cuskley et al. (2018) used 28
n_tokens = 250  # int: number of tokens in vocabulary. Cuskley et al. seem to h
n_inflections = 6  # int: number of inflections. Cuskley et al. (2018) used 12
zipf_exponent = 2  # int: exponent used to create Zipfian frequency distributio
k_proficiency = 250  # int: token threshold that determines proficiency. Cuskle
r_replacement = 0.001  # float: replacement rate for turnover condition. Cuskle
# At every interaction, there is an r chance that a randomly selected learner w
g_growth = 0.001  # float: growth rate for growth condition. Cuskley et al. (20
# At every interaction, there's a g chance that a new learner will be *added* t
replacement = True  # Boolean: determines whether this simulation includes repl
growth = False  # Boolean; determines whether this simulation includes growth
t_timesteps = 5000  # int: number of timesteps to run per simulation. Cuskley e
d_memory = 100  # int: no. of timesteps after which agent forgets lemma-inflect
```

We start with a function that can create a vocabulary array that contains n_tokens tokens of n_lemmas types, with a Zipfian frequency distribution.

Skim the function and check whether you understand what it's doing and why.

In [ ]:
```
def generate_vocab(n_lemmas, zipf_exponent, n_tokens):
        """
        Generates a vocabulary (numpy array of n_tokens tokens of n_lemmas type
        :param n_lemmas: int: number of lemmas
        :param zipf_exponent: int: exponent used to create Zipfian frequency di
        :param n_tokens: int: number of tokens in vocabulary. Cuskley et al. se
        :return: (1) numpy array containing n_tokens tokens of n_lemmas types;
        """
        lemma_indices = np.arange(n_lemmas)  # create numpy array with index fo
        zipf_dist = np.random.zipf(zipf_exponent, size=n_lemmas)  # create Zipf
        zipf_dist_in_probs = np.divide(zipf_dist, np.sum(zipf_dist))
        zipf_dist_for_n_tokens = np.multiply(zipf_dist_in_probs, n_tokens)
        zipf_dist_for_n_tokens = np.ceil(zipf_dist_for_n_tokens)  # Round UP, s
        vocabulary = np.array([])
        for i in range(len(lemma_indices)):
                lemma_index = lemma_indices[i]
                lemma_freq = zipf_dist_for_n_tokens[i]
                vocabulary = np.concatenate((vocabulary, np.array([lemma_index
        for j in range(2):  # doing this twice because sth weird w/ np.delete()
                # (possibly to do with later index going out of bounds once pre
                if vocabulary.shape[0] > n_tokens:  # if vocab is larger than n
                        random_indices = np.random.choice(np.arange(vocabulary.
                        vocabulary = np.delete(vocabulary, random_indices)
        np.random.shuffle(vocabulary)  # finally, shuffle the array so that tok
        vocabulary = vocabulary.astype(int)
        return vocabulary, np.log(zipf_dist_in_probs)
```

The rest of the code is divided up into the following four classes: (In object-oriented programming languages like Python, a class is essentially a collection of attributes and functions that belong together.)

- `Inflection`
- `Lemma`
- `Agent`
- `Simulation`

Inflection and Lemma classes:

We start with the Inflection class. This class defines an inflection as paired with a lemma.

Skim the Inflection class and check whether you understand what it's doing and why.

In [ ]:
```
class Inflection:
        """
        Class which defines an inflection as paired with a lemma
        """
        def __init__(self, interactions=0, successes=0, weight=np.nan, last_int
                """
                Initialises Inflection object
                :param interactions: int: number of interactions agents has had
```

```
            :param successes: int: no. of successful interactions agent has
            :param weight: float: no. of successes / no. of interactions. I
            :param last_interaction: int: timestep when the pairing was las
            """
            self.interactions = interactions
            self.successes = successes
            self.weight = weight
            self.last_interaction = last_interaction

    def empty_inflection(self):
            """
            Empties the inflection by resetting each of its attributes; use
            (i.e., d_memory timesteps) has elapsed since last interaction w
            :return: resets each of the inflection object's attributes; doe
            """
            self.interactions = 0
            self.successes = 0
            self.weight = np.nan
            self.last_interaction = np.nan
```

Next is the Lemma class, which can update the inflections of a given lemma depending on the outcomes of interactions between agents:

In [ ]:

```
class Lemma:
    """
    Lemma class
    """
    def __init__(self, lemma_index, tokens, seen, inflections):
            """
            Initialises Lemma object
            :param lemma_index: int: index of the lemma
            :param tokens: int: number of times the agent has encountered t
            :param seen: Boolean: whether the agent has encountered this le
            :param inflections: dictionary with keys: "interactions", "succ
            """
            self.index = lemma_index
            self.tokens = tokens
            self.seen = seen
            self.inflections = inflections

    def reset_lemma(self):
            """
            Initialises/resets all attributes of the lemma object
            :param self:
            :return:
            """
            self.tokens = 0
            self.seen = False
            self.inflections = [Inflection() for i in range(n_inflections)]

    def add_inflection(self, infl_index, outcome, timestep):
            """
            Adds an inflection to the lemma (as a result of an interaction
            with weight depending on the outcome of the interaction (succes
            previous interaction in which this inflection was used
```

```
        :param infl_index: int: index of the inflection in self.inflect
        :param outcome: int: 1 if success (i.e., if receiver has lemma-
        :param timestep: int: timestep of current interaction
        :return: updates attributes of lemma object; doesn't return any
        """
        self.seen = True
        self.tokens = 1
        self.inflections[infl_index].interactions = 1
        self.inflections[infl_index].successes = outcome
        self.inflections[infl_index].weight = float(outcome) / float(se
        self.inflections[infl_index].last_interaction = timestep

    def update_inflection(self, infl_index, outcome, timestep):
        """
        Updates a lemma-inflection pairing based on the outcome of an i
        :param infl_index: int: index of the inflection in self.inflect
        :param outcome: int: 1 if success (i.e., if receiver has lemma-
        :param timestep: int: timestep of current interaction
        :return: updates attributes of lemma object; doesn't return any
        """
        self.tokens += 1
        self.inflections[infl_index].interactions += 1
        self.inflections[infl_index].successes += outcome
        self.inflections[infl_index].weight = float(self.inflections[in
                self.inflections[infl_index].interactions)
        self.inflections[infl_index].last_interaction = timestep

    def has_inflection(self, infl_index):
        """
        Checks whether agent already has a specific inflection (indicat
        :param infl_index: int: index of the inflection in self.inflect
        :return: Boolean: True if agent already has this specific infle
        """
        if self.inflections[infl_index].interactions > 0:
                return True
        else:
                return False

    def get_best(self):
        """
        Finds indices of inflections with highest weight for this lemma
        :return: int: index of highest-weighted inflection. If multiple
        """
        weight_array = np.array([self.inflections[i].weight for i in ra
        if np.isnan(weight_array).all() == True:  # if only NANs in the
                max_index = np.random.choice(np.arange(
                        len(self.inflections)))
        else:
                max_weight = np.nanmax(weight_array)
                max_indices = np.where(weight_array == max_weight)[0]
                max_index = np.random.choice(max_indices)
        return max_index

    def has_any_inflection(self):
        """
        Checks whether this lemma has any inflections yet (this is cons
        inflections have come up in an interaction about this lemma bet
```

```
            :return: Boolean: False if lemma object doesn't have any inflec
            """
            interactions_per_inflection = np.array([self.inflections[i].int
            if np.sum(interactions_per_inflection) == 0:
                    return False
            elif np.sum(interactions_per_inflection) > 0:
                    return True

    def purge(self, timestep):
            """
            Resets lemma-inflection pairing if memory window (d_memory) has
            :param timestep: int: timestep of current interaction
            :return: updates self.inflections attribute of lemma object; do
            """
            for i in range(len(self.inflections)):
                    if (timestep - self.inflections[
                            i].last_interaction) > d_memory:  # d_memory is
                            self.inflections[i].empty_inflection()
```

Exercise 1:

In this exercise, we're going to have a look at what a Lemma object looks like, and how it can be used.

In order to initialise a Lemma object, we have to specify several input arguments:

- `lemma_index`
- `tokens`
- `seen`
- `inflections`

The values of these attributes don't really matter for the purposes of the current exercise, so you can just initialise your Lemma object with random values.

Note that the `inflections` input argument expects a list of Inflection objects. To create these, you can do:

`[Inflection() for i in range(n_inflections)]`

This creates a list of Inflection objects with default values.

To print the attributes of an object, you can use:

`print(object_name.__dict__)`

So, for example, assuming you have created a Lemma object named `my_lemma`, you can do: `print(my_lemma.__dict__)` in order to inspect it.

a) Write code that does the following:

- Create a Lemma object

- write a for-loop that uses the Lemma's .update_inflection() method 10 times, where at each timestep:
  - an inflection index is selected randomly from range n_inflections
  - an outcome value is selected randomly from the options [0, 1] (representing failure and success, respectively)
  - the Lemma object's .update_inflection() method is called with the input arguments generated above (timestep is not relevant for this exercise, but you can just set it to the index of your for-loop, if you feel like it)
  - inspect the Lemma object and how it changes. This requires not just printing the lemma object, but also printing each of the Inflection objects in the lemma's self.inflections attribute (again using print(object_name.**dict**))

b) How is the weight of a given inflection calculated each time the .update_inflection() method is called?

In [ ]:
```
lemma_index = 1
tokens = 1
seen = False
inflections = [Inflection() for i in range(n_inflections)]

my_lemma = Lemma(lemma_index, tokens, seen, inflections)
print(my_lemma.__dict__)


for i in range(1,10):
    print("")
    infl_index = np.random.choice(range(1, n_inflections))
    outcome = np.random.choice([0,1])
    timestep = i
    my_lemma.update_inflection(infl_index, outcome, timestep)
    for inflection in my_lemma.inflections:
        print(inflection.__dict__)
    print(f"infl_index: {infl_index};   outcome: {outcome};  timestep: {timeste
```

```
{'index': 1, 'tokens': 1, 'seen': False, 'inflections': [<__main__.Inflection
object at 0x13eb8e1c0>, <__main__.Inflection object at 0x13eb428e0>, <__main_
_.Inflection object at 0x13eb42b80>, <__main__.Inflection object at 0x13eb4258
0>, <__main__.Inflection object at 0x13eb42c10>, <__main__.Inflection object a
t 0x13eb427f0>]}

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
infl_index: 3;   outcome: 0;  timestep: 1

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 1, 'weight': 1.0, 'last_interaction': 2}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
infl_index: 1;   outcome: 1;  timestep: 2

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 1, 'weight': 1.0, 'last_interaction': 2}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 3}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
infl_index: 4;   outcome: 0;  timestep: 3

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 2, 'successes': 1, 'weight': 0.5, 'last_interaction': 4}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 3}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
infl_index: 1;   outcome: 0;  timestep: 4

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 2, 'successes': 1, 'weight': 0.5, 'last_interaction': 4}
{'interactions': 1, 'successes': 1, 'weight': 1.0, 'last_interaction': 5}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 3}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
infl_index: 2;   outcome: 1;  timestep: 5

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 3, 'successes': 2, 'weight': 0.6666666666666666, 'last_intera
ction': 6}
{'interactions': 1, 'successes': 1, 'weight': 1.0, 'last_interaction': 5}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 3}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
infl_index: 1;   outcome: 1;  timestep: 6

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 3, 'successes': 2, 'weight': 0.6666666666666666, 'last_intera
```

```
ction': 6}
{'interactions': 2, 'successes': 2, 'weight': 1.0, 'last_interaction': 7}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 3}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
infl_index: 2;   outcome: 1;  timestep: 7

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 3, 'successes': 2, 'weight': 0.6666666666666666, 'last_intera
ction': 6}
{'interactions': 2, 'successes': 2, 'weight': 1.0, 'last_interaction': 7}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 3}
{'interactions': 1, 'successes': 1, 'weight': 1.0, 'last_interaction': 8}
infl_index: 5;   outcome: 1;  timestep: 8

{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 3, 'successes': 2, 'weight': 0.6666666666666666, 'last_intera
ction': 6}
{'interactions': 2, 'successes': 2, 'weight': 1.0, 'last_interaction': 7}
{'interactions': 2, 'successes': 0, 'weight': 0.0, 'last_interaction': 9}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 3}
{'interactions': 1, 'successes': 1, 'weight': 1.0, 'last_interaction': 8}
infl_index: 3;   outcome: 0;  timestep: 9
```

The weight is calculated based on the following formula:
weight = number of successes for the inflection / n of interactions for the inflection

Agent class:

Next, we have the Agent class, which contains all the attributes/properties of an agent (e.g., their vocabulary, how many tokens they've seen in their life so far, whether that makes them a type-generaliser or a token-generaliser, etc.), and all the functions (named "methods" if we're talking about a class) that allow the agent to interact and update their attributes/properties based on the content and outcome of that interaction.

In [ ]:
```
class Agent:
        """
        Agent class
        """
        def __init__(self, tokens=0, k_threshold=k_proficiency, memory_window=c
                                 is_active=False):
                """
                Initialises Agent object
                :param tokens: int: number of tokens seen by the agent in total
                :param k_threshold: int: token threshold that determines profic
                :param memory_window: int: no. of timesteps after which agent f
                :param type_generalise: Boolean: False = agent is token-general
                :param is_active: Boolean: Initial value: False. Agent's status
                """
                self.tokens = tokens
                self.k_threshold = k_threshold
                self.memory_window = memory_window
```

```
        self.type_generalise = type_generalise
        self.is_active = is_active
        empty_inflections = [Inflection() for i in range(n_inflections)
        self.vocabulary = [Lemma(0, 0, False, empty_inflections) for x

    def reset_agent(self):
        """
        Resets agent's attributes to initial/empty
        :return: resets agent's attributes; doesn't return anything
        """
        self.is_active = True
        self.tokens = 0
        self.type_generalise = False
        for lemma in self.vocabulary:
            lemma.reset_lemma()

    def has_inflections(self, lemma_index):
        """
        Checks whether agent has any inflections for a particular lemma
        :param lemma_index: int: index of particular Lemma object in se
        :return: Boolean: True if agent has any inflections for this pa
        """
        return self.vocabulary[lemma_index].has_any_inflection()

    def update_lemma(self, lemma_index, infl_index, outcome, timestep):
        """
        Update the entry for a particular lemma
        :param lemma_index: int: index of the lemma (in the agent's sel
        :param infl_index: int: index of the inflection
        :param outcome: int: 1 if success (i.e., if receiver has lemma-
        :param timestep: int: timestep of current interaction
        :return: updates lemma in agent's vocabulary; doesn't return an
        """
        self.tokens += 1
        # If lemma-inflection pairing exists, update the weighting acco
        if self.vocabulary[lemma_index].has_inflection(infl_index):
            self.vocabulary[lemma_index].update_inflection(infl_inc
        # If lemma-inflection pairing doesn't exist yet, create it:
        else:
            self.vocabulary[lemma_index].add_inflection(infl_index,
        # Purge the inflections of the lemma (i.e., remove inflections
        self.vocabulary[lemma_index].purge(timestep)
        # Finally, set agent's self.type_generalise attribute depending
        if self.tokens > self.k_threshold:
            self.type_generalise = True
        else:
            self.type_generalise = False

    def get_best(self, lemma_index):
        """
        Get best (i.e., heighest-weighted) inflection for this lemma
        :param lemma_index: int: index of the lemma (in the agent's sel
        :return: int: index of best (i.e., heighest-weighted) inflectio
        """
        return self.vocabulary[lemma_index].get_best()

    def get_token_best(self):
```

```
        """
        Token-generalise: Look across vocab and extend rule that was us
        :return: int: index of inflection used across most *tokens*
        """
        max_tokens = np.zeros(n_inflections)
        for lemma_index in range(len(self.vocabulary)):
                for i in range(n_inflections):
                        max_tokens[i] += self.vocabulary[lemma_index].i
        max_successes = np.amax(max_tokens)
        max_token_indices = np.where(max_tokens == max_successes)[0]
        max_index = np.random.choice(max_token_indices)
        return max_index

    def get_type_best(self):
        """
        Type-generalise: Look across vocab and extend the rule which ap
        :return: int: index of inflection used across most *types*
        """
        max_types = np.zeros(n_inflections)
        for lemma_index in range(len(self.vocabulary)):
                best_inflection = self.vocabulary[lemma_index].get_best
                max_types[best_inflection] += 1
        max_values = np.amax(max_types)
        max_token_indices = np.where(max_types == max_values)[0]
        max_index = np.random.choice(max_token_indices)
        return max_index

    def generate_inflection(self):
        """
        If a lemma has no inflections, generate an inflection based on
        :return: int: index of newly generated (/generalised) inflectio
        """
        inflection_utterance = np.nan
        # If self.type_generalise is True (= when agent has exceeded k_
        if self.type_generalise:
                inflection_utterance = self.get_type_best()
                # If preferred generalisation process doesn't provide i
                if np.isnan(inflection_utterance):
                        inflection_utterance = self.get_token_best()
        # If self.type_generalise is False (=agent hasn't reached k_thr
        else:
                inflection_utterance = self.get_token_best()
                # If preferred generalisation process doesn't provide i
                if np.isnan(inflection_utterance):
                        inflection_utterance = self.get_type_best()
        # If agent has no inflections in vocabulary, they will choose a
        if np.isnan(inflection_utterance):
                inflection_utterance = np.random.choice(np.arange(n_inf
        return inflection_utterance

    def receive(self, lemma_index, infl_index, timestep):
        """
        Take inflection in as receiver and update lemmas in vocabulary
        :param lemma_index: int: index of the lemma (in the agent's sel
        :param infl_index: int: index of the inflection
        :param timestep: int: timestep of current interaction
        :return: Boolean: 1 if interaction is success (= lemma-inflecti
```

```
        """
        # If agent has any inflections for this lemma:
        if self.has_inflections(lemma_index):
                # If the agent has this particular inflection for this
                if self.vocabulary[lemma_index].has_inflection(infl_inc
                        self.update_lemma(lemma_index, infl_index, 1, t
                        return 1
                else:
                        self.update_lemma(lemma_index, infl_index, 0, t
                        return 0
        # If agent doesn't have any inflections for this lemma, generat
        else:
                guess = self.generate_inflection()
                # If the newly generated inflection matches the inflect
                if guess == infl_index:
                        self.update_lemma(lemma_index, infl_index, 1, t
                        return 1
                else:
                        self.update_lemma(lemma_index, infl_index, 0, t
                        return 0
```

Exercise 2:

In this exercise, we're going to create two Agent objects and have them interact with each other.

When you create an Agent object, you *can* specify a number of input arguments. However, the Agent class is defined in such a way that each of these input arguments has a default value. This means that you can simply create a "default" Agent using:

`my_agent = Agent()`

a) Write code to do the following:

- Create two agent objects called `agent_1` and `agent_2`
- Write a for-loop to loop through the following steps 10 times:
  - Randomly choose a lemma from range(n_lemmas), and print the chosen lemma
  - Randomly assign the role of producer to one of the agents, and the role of receiver to the other agent
  - Make the producer agent produce an utterance (i.e., an inflection for the lemma that was chosen above) using the Agent's `.get_best()` method, and print the utterance
  - Print the values in the receiver's vocabulary for the lemma-inflection pairing that was uttered, *before* running the .receive method
  - Make the receiver agent update their vocabulary based on the utterance received, using the Agent's `.receive()` method. Save the outcome of this interaction (= output of the `.receive()` method) in a variable and print it
  - Print the values in the receiver's vocabulary for the lemma-inflection pairing that was uttered, *after* running the .receive method, and inspect how they've changed.

b) Do you notice any pattern in which inflections the agents use across the different lemmas? If so, try to explain this pattern.

In [ ]:
```
agent_1 = Agent()
agent_2 = Agent()

for i in range(1,10):
    print("")
    agent_rd = np.random.choice([1,2])
    if agent_rd == 1:
        producer = agent_1
        receiver = agent_2
    else:
        producer = agent_2
        receiver = agent_1
    print(f"[The producer is agent {agent_rd}]==================")

    lemma_index = np.random.choice(n_lemmas)
    print(f"lemma_index: {lemma_index}")

    utterance = producer.get_best(lemma_index)
    print(f"Utterance (inflection): {utterance}")
    print("Receiver's vocab before .receive()")
    receiver_vocab = receiver.vocabulary
    for inflection in receiver_vocab[lemma_index].inflections:
        print(inflection.__dict__)

    receiver.receive(lemma_index, utterance, i)
    print("")
    print("Receiver's vocab after .receive()")
    for inflection in receiver_vocab[lemma_index].inflections:
        print(inflection.__dict__)
```

```
[The producer is agent 1]==================
lemma_index: 12
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

[The producer is agent 2]==================
lemma_index: 11
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 2}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

[The producer is agent 2]==================
lemma_index: 7
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 2}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 2, 'successes': 1, 'weight': 0.5, 'last_interaction': 3}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
```

```
[The producer is agent 2]==================
lemma_index: 10
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 2, 'successes': 1, 'weight': 0.5, 'last_interaction': 3}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 3, 'successes': 2, 'weight': 0.6666666666666666, 'last_intera
ction': 4}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

[The producer is agent 2]==================
lemma_index: 0
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 3, 'successes': 2, 'weight': 0.6666666666666666, 'last_intera
ction': 4}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 4, 'successes': 3, 'weight': 0.75, 'last_interaction': 5}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

[The producer is agent 1]==================
lemma_index: 9
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 1, 'successes': 0, 'weight': 0.0, 'last_interaction': 1}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 2, 'successes': 1, 'weight': 0.5, 'last_interaction': 6}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
```

```
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

[The producer is agent 2]==================
lemma_index: 5
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 4, 'successes': 3, 'weight': 0.75, 'last_interaction': 5}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 5, 'successes': 4, 'weight': 0.8, 'last_interaction': 7}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

[The producer is agent 2]==================
lemma_index: 7
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 5, 'successes': 4, 'weight': 0.8, 'last_interaction': 7}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 6, 'successes': 5, 'weight': 0.8333333333333334, 'last_intera
ction': 8}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

[The producer is agent 2]==================
lemma_index: 7
Utterance (inflection): 2
Receiver's vocab before .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 6, 'successes': 5, 'weight': 0.8333333333333334, 'last_intera
ction': 8}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}

Receiver's vocab after .receive()
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 7, 'successes': 6, 'weight': 0.8571428571428571, 'last_intera
```

```
ction': 9}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
{'interactions': 0, 'successes': 0, 'weight': nan, 'last_interaction': nan}
```

I don't know if I read the output correctly, but the producer seens to produce utterances based on the inflections with highest weight across lemmas, not for the particular lemma. I thought the get_best() function produces an inflection with highest weight for the lemma if the producer has already encountered the lemma or a random inflection when they haven't. In fact, for instance, the number of interactions in the third interaction should be 0 for all the inflections because they have never encounted the lemma "7" before, but it looks like they know that they have encountered the inflection for other lemmas but don't know with which lemma they encountered the inflection.

Simulation class:

That brings us to the final class: Simulation. This class allows us to run a batch of simulation runs given the parameter settings specified at the top of the notebook.

In [ ]:
```
class Simulation:
        """
        Simulation class
        """
        def __init__(self, pop_size):
                """
                Initialises simulation object with self.pop_size, self.populati
                :param pop_size: int: population size
                """
                self.pop_size = pop_size
                self.population = [Agent() for x in range(3000)]  # Create init
                self.running_popsize = pop_size  # int: keeps track of the char
                self.n_interactions = pop_size  # int: no. of interactions per
                self.vocabulary, self.log_freqs_per_lemma = generate_vocab(n_le
                self.all_tokens = 0  # Keeps track of total number of tokens th
                self.global_inflections = np.zeros(n_inflections)  # Keeps trac
                self.global_counts = np.zeros(n_lemmas)  # Keeps track of the i
                self.pop_size_column = np.zeros(n_runs*t_timesteps*n_lemmas)
                self.r_column = np.zeros(n_runs*t_timesteps*n_lemmas)
                self.tstep_column = np.zeros(n_runs*t_timesteps*n_lemmas)
                self.lemma_column = np.zeros(n_runs*t_timesteps*n_lemmas)
                self.log_freq_column = np.zeros(n_runs * t_timesteps * n_lemmas
                self.infl_column = np.zeros(n_runs*t_timesteps*n_lemmas)
                self.vocab_entropy_column = np.zeros(n_runs*t_timesteps*n_lemma
                self.meaning_entropy_column = np.zeros(n_runs*t_timesteps*n_lem

        def interaction(self, producer, receiver, lemma, current_timestep):
                """
                Run single interaction between producer and receiver
                :param producer: int: index of producer agent in self.populatic
                :param receiver: int: index of receiver agent in self.populatic
                :param lemma: int: index of lemma in agent.vocabulary
                :param current_timestep: int: current timestep
                :return: updates the lemma in the producer's and receiver's voc
```

```
        and self.global_inflections, which keeps track of frequency of
        doesn't return anything
        """
        if self.population[producer].has_inflections(lemma):
                utterance = self.population[producer].get_best(lemma)
                result = self.population[receiver].receive(lemma, utter
        else:
                utterance = self.population[producer].generate_inflecti
                result = self.population[receiver].receive(lemma, utter
        self.population[producer].update_lemma(lemma, utterance, result
        self.global_inflections[utterance] += 1

def replace_agent(self):
        """
        Replace an agent in turnover condition. Randomly selects an age
        (equivalent to removing the selected agent and adding a new age
        :return: updates self.population by resetting the attributes of
        """
        # Generate random float from uniform dist. [0.0, 1.0); if float
        if np.random.random() <= r_replacement:
                chosen_one_index = np.random.choice(np.arange(self.runn
                self.population[chosen_one_index].reset_agent()

def add_agent(self):
        """
        Add agent to population in growth condition by setting one of t
        :return: updates self.population by adding a new agent (by sett
        """
        if np.random.random() <= g_growth:
                self.running_popsize += 1
                # Take next of "dormant" agents in line and turn its .i
                self.population[self.running_popsize-1].is_active = Tru

def timestep(self, current_timestep):
        """
        Runs through 1 timestep in simulation. Each timestep consists o
        Cuskley et al. (2018) used n_interactions = pop_size
        :param current_timestep: int: current timestep
        :return: Updates attributes of population and its agents based
        and whether the replacement and growth conditions are turned on
        """
        vocab_index = 0
        for i in range(self.n_interactions):
                # Randomly select producer and receiver agent:
                producer_index = np.random.choice(np.arange(self.runnin
                receiver_index = np.random.choice(np.arange(self.runnin
                # Make sure producer and receiver are not the same agen
                while producer_index == receiver_index:
                        receiver_index = np.random.choice(np.arange(sel
                # If we've reached the end of the vocabulary array, re-
                if vocab_index >= (n_tokens-1):
                        np.random.shuffle(self.vocabulary)
                        vocab_index = 0
                topic = self.vocabulary[vocab_index]
                self.interaction(producer_index, receiver_index, topic,
                if growth:  # growth is global variable (Boolean)
                        self.add_agent()
```

```
            if replacement:  # growth is global variable (Boolean)
                    self.replace_agent()
            self.global_counts[topic] += 1
            self.all_tokens += 1
            vocab_index += 1

    def inflections_in_vocab(self):
        """
        Counts total number of inflections present in population
        :return: int: total number of inflections present in populatior
        """
        infl_counts = np.zeros(n_inflections)
        total_inflections = 0.
        # First, create array which counts for each inflection how many
        for l in range(n_lemmas):
            for a in range(self.running_popsize):
                if self.population[a].has_inflections(l):
                    best_infl = self.population[a].get_best
                    infl_counts[best_infl] += 1
        # Then, get the total number of inflections which has a count >
        for i in range(n_inflections):
            if infl_counts[i] > 0:
                total_inflections += 1
        return total_inflections

    def get_entropy(self, probability_array):
        """
        Calculates the entropy from a list of probabilities/frequencies
        :param probability_array: 1D numpy array containing probabilit:
        :return: float: entropy
        """
        entropy = 0.
        for p in probability_array:
            if p > 0.:
                entropy += p * np.log2(1./p)
        return entropy

    def vocabulary_entropy(self):
        """
        Calculates entropy of inflection across the vocabulary, H_v
        :return: float: H_v
        """
        # how predictable is the inflection of any given lemma?
        # for each lemma
        inflection_probs = np.zeros(n_inflections)
        denominator = 0.
        for l in range(n_lemmas):
            for a in range(self.running_popsize):
                if self.population[a].vocabulary[l].has_any_inf
                    denominator += 1
                    best_infl = self.population[a].get_best
                    inflection_probs[best_infl] += 1
        inflection_probs = np.divide(inflection_probs, denominator)
        return self.get_entropy(inflection_probs)

    def meaning_entropy(self, lemma):
        """
```

```
        Calculates entropy of the inflection for a specific lemma, H_l
        :param lemma: int: index of lemma that should be conditioned or
        :return: float: H_l
        """
        # what is the probability of each inflection given this lemma?
        inflections = np.zeros(n_inflections)
        lemma_count = 0.
        for a in range(self.running_popsize):
                if self.population[a].has_inflections(lemma):
                        best_infl = self.population[a].get_best(lemma)
                        inflections[best_infl] += 1.
                        lemma_count += 1.
        inflection_probs = np.divide(inflections, lemma_count)
        return self.get_entropy(inflection_probs)

def single_run(self, run_number, counter):
        """
        Runs a single simulation. Each run is t_timesteps long (Cuskley
        :param run_number: int: index of current run
        :return: Updates the Simulation object's attributes (specifical
        """
        for t in range(t_timesteps):
                if t % 500 == 0:  # after every 50 timesteps, print the
                        print("t: "+str(t))
                self.timestep(t)
                total_inflections = self.inflections_in_vocab()
                if t == t_timesteps-1:
                        vocab_entropy = self.vocabulary_entropy()
                for lemma_index in range(n_lemmas):
                        self.pop_size_column[counter] = self.pop_size
                        self.r_column[counter] = run_number
                        self.tstep_column[counter] = t
                        self.lemma_column[counter] = lemma_index
                        self.log_freq_column[counter] = self.log_freqs_
                        self.infl_column[counter] = total_inflections
                        if t == t_timesteps-1:
                                self.vocab_entropy_column[counter] = vo
                                self.meaning_entropy_column[counter] =
                        else:
                                self.vocab_entropy_column[counter] = np
                                self.meaning_entropy_column[counter] =
                        counter += 1
        print("self.running_popsize at end of simulation:")
        print(self.running_popsize)
        return counter

def multi_runs(self):
        """
        Runs multiple runs of the simulation
        :return: pandas dataframe containing all results
        """
        for i in range(self.pop_size):
                self.population[i].is_active = True
        counter = 0
        for r in range(n_runs):
                print('')
                print("r: "+str(r))
```

```
                # First, reset self.all_tokens, self.global_inflections
                self.all_tokens = 0
                self.global_inflections = np.zeros(n_inflections)
                self.global_counts = np.zeros(n_lemmas)
                # Then, run a new run:
                counter = self.single_run(r, counter)
        # After all runs have finished, turn the numpy arrays with resu
        results_dict = {"pop_size": self.pop_size_column,
                        "run": self.r_column,
                        "timestep": self.tstep_column,
                        "lemma": self.lemma_column,
                        "log_freq": self.log_freq_colum
                        "n_inflections": self.infl_colu
                        "vocab_entropy": self.vocab_ent
                        "meaning_entropy": self.meaning
        results_dataframe = pd.DataFrame(results_dict)
        return results_dataframe
```

Final functions for running simulations and plotting results

We have now seen all the classes that the code for this model consists of. In addition to those, we also need a function that can run simulations for several population sizes, and combine their results into one big dataframe, which also gets saved as a pickle file (to your current working directory).

In [ ]:
```
def run_multi_sizes(pop_sizes):
        """
        Runs simulations for each pop_size in pop_sizes
        :param pop_sizes: list of ints specifying the different population size
        :return: pandas dataframe containing simulation results for all pop_siz
        """
        start_time = time.time()
        frames = []
        # First run simulations for each of the pop_sizes:
        for pop_size in pop_sizes:
                print('')
                print("pop_size is:")
                print(pop_size)
                simulation = Simulation(pop_size)
                results_dataframe = simulation.multi_runs()
                frames.append(results_dataframe)
                print("Simulation(s) took %s minutes to run" % round(((time.tim
        # Then combine the results for each of the pop_sizes into one big data
        combined_dataframe = pd.concat(frames, ignore_index=True)
        combined_dataframe.to_pickle("./results_"+"n_runs_"+str(n_runs)+"_tstep
        return combined_dataframe
```

Finally, we need a couple of plotting functions. The resulting plots also get saved as pdfs.

In [ ]:
```
def plot_vocab_entropy(results_df): #Figure A
        sns.set_style("darkgrid")
        with sns.color_palette("deep", 2):
```

```
                sns.displot(data=results_df, x="vocab_entropy", hue="pop_size",
        plt.savefig("./Hv_plot_"+"n_runs_"+str(n_runs)+"_tsteps_" + str(t_times


def plot_meaning_entropy_by_freq(results_df): #Figure B
        sns.set_style("darkgrid")
        with sns.color_palette("deep", 2):
                sns.lineplot(data=results_df, x="log_freq", y="meaning_entropy'
        plt.savefig("./Hl_plot_"+"n_runs_"+str(n_runs)+"_tsteps_" + str(t_times


def plot_active_inflections_over_time(results_df): #Figure C
        sns.set_style("darkgrid")
        with sns.color_palette("deep", 2):
                sns.lineplot(data=results_df, x="timestep", y="n_inflections",
        plt.savefig("./Inflections_plot_"+"n_runs_"+str(n_runs)+"_tsteps_" + st
```

Exercise 3:

Have a look at each of the plotting functions above, and which measures from the results_dataframe they visualise.

Are all measures that are reported in the figures in Cuskley et al. (2018) represented here? If not, explain what is missing, and what that missing measure captures.

Figure (D), which captures the probability of a given lemma having its most common inflection measured across the population be the type-dominant “regular” inflection, is missing from above.

Example of how to run simulation:

Below is an example of how to run a simulation for each population size in the pop_sizes parameter, and to save and print the resulting dataframe. **Note** that the code cell below temporarily decreases the number of timesteps (as defined by the `t_timesteps` parameter at the top of the code), because the code cell below is just meant as a quick example.

In [ ]:
```
t_timesteps = 100

results_dataframe = run_multi_sizes(pop_sizes)
print('')
print("results_dataframe is:")
print(results_dataframe)
```

```
pop_size is:
20

r: 0
t: 0
self.running_popsize at end of simulation:
20

r: 1
t: 0
self.running_popsize at end of simulation:
20
Simulation(s) took 0.06 minutes to run

pop_size is:
100

r: 0
t: 0
self.running_popsize at end of simulation:
100

r: 1
t: 0
self.running_popsize at end of simulation:
100
Simulation(s) took 0.33 minutes to run

results_dataframe is:
      pop_size  run  timestep  lemma  log_freq  n_inflections  vocab_entropy
\
0         20.0  0.0       0.0    0.0 -2.602690            6.0            NaN
1         20.0  0.0       0.0    1.0 -3.295837            6.0            NaN
2         20.0  0.0       0.0    2.0 -2.197225            6.0            NaN
3         20.0  0.0       0.0    3.0 -3.295837            6.0            NaN
4         20.0  0.0       0.0    4.0 -2.602690            6.0            NaN
...        ...  ...       ...    ...       ...            ...            ...
5595     100.0  1.0      99.0    9.0 -3.465736            4.0       0.090465
5596     100.0  1.0      99.0   10.0 -3.465736            4.0       0.090465
5597     100.0  1.0      99.0   11.0 -3.465736            4.0       0.090465
5598     100.0  1.0      99.0   12.0 -3.465736            4.0       0.090465
5599     100.0  1.0      99.0   13.0 -1.856298            4.0       0.090465

      meaning_entropy
0                 NaN
1                 NaN
2                 NaN
3                 NaN
4                 NaN
...               ...
5595         0.082837
5596         0.000000
5597         0.084262
5598         0.082143
5599         0.081462

[5600 rows x 8 columns]
```

The simulation results have been saved in a Pandas dataframe. To extract only the results of the final timestep, for example, you can do the following:

```
In [ ]: final_timestep_results = results_dataframe[results_dataframe["timestep"]==t_tin
        print("final_timestep_results are:")
        print(final_timestep_results)
```

```
final_timestep_results are:
      pop_size  run  timestep  lemma  log_freq  n_inflections  vocab_entropy  \
1386      20.0  0.0      99.0    0.0 -2.602690            2.0       0.703225
1387      20.0  0.0      99.0    1.0 -3.295837            2.0       0.703225
1388      20.0  0.0      99.0    2.0 -2.197225            2.0       0.703225
1389      20.0  0.0      99.0    3.0 -3.295837            2.0       0.703225
1390      20.0  0.0      99.0    4.0 -2.602690            2.0       0.703225
1391      20.0  0.0      99.0    5.0 -3.295837            2.0       0.703225
1392      20.0  0.0      99.0    6.0 -1.909543            2.0       0.703225
1393      20.0  0.0      99.0    7.0 -2.197225            2.0       0.703225
1394      20.0  0.0      99.0    8.0 -3.295837            2.0       0.703225
1395      20.0  0.0      99.0    9.0 -1.909543            2.0       0.703225
1396      20.0  0.0      99.0   10.0 -2.602690            2.0       0.703225
1397      20.0  0.0      99.0   11.0 -3.295837            2.0       0.703225
1398      20.0  0.0      99.0   12.0 -3.295837            2.0       0.703225
1399      20.0  0.0      99.0   13.0 -3.295837            2.0       0.703225
2786      20.0  1.0      99.0    0.0 -2.602690            2.0       0.498028
2787      20.0  1.0      99.0    1.0 -3.295837            2.0       0.498028
2788      20.0  1.0      99.0    2.0 -2.197225            2.0       0.498028
2789      20.0  1.0      99.0    3.0 -3.295837            2.0       0.498028
2790      20.0  1.0      99.0    4.0 -2.602690            2.0       0.498028
2791      20.0  1.0      99.0    5.0 -3.295837            2.0       0.498028
2792      20.0  1.0      99.0    6.0 -1.909543            2.0       0.498028
2793      20.0  1.0      99.0    7.0 -2.197225            2.0       0.498028
2794      20.0  1.0      99.0    8.0 -3.295837            2.0       0.498028
2795      20.0  1.0      99.0    9.0 -1.909543            2.0       0.498028
2796      20.0  1.0      99.0   10.0 -2.602690            2.0       0.498028
2797      20.0  1.0      99.0   11.0 -3.295837            2.0       0.498028
2798      20.0  1.0      99.0   12.0 -3.295837            2.0       0.498028
2799      20.0  1.0      99.0   13.0 -3.295837            2.0       0.498028
4186     100.0  0.0      99.0    0.0 -1.519826            4.0       0.045955
4187     100.0  0.0      99.0    1.0 -3.465736            4.0       0.045955
4188     100.0  0.0      99.0    2.0 -3.465736            4.0       0.045955
4189     100.0  0.0      99.0    3.0 -1.268511            4.0       0.045955
4190     100.0  0.0      99.0    4.0 -3.465736            4.0       0.045955
4191     100.0  0.0      99.0    5.0 -3.465736            4.0       0.045955
4192     100.0  0.0      99.0    6.0 -3.465736            4.0       0.045955
4193     100.0  0.0      99.0    7.0 -3.465736            4.0       0.045955
4194     100.0  0.0      99.0    8.0 -3.465736            4.0       0.045955
4195     100.0  0.0      99.0    9.0 -3.465736            4.0       0.045955
4196     100.0  0.0      99.0   10.0 -3.465736            4.0       0.045955
4197     100.0  0.0      99.0   11.0 -3.465736            4.0       0.045955
4198     100.0  0.0      99.0   12.0 -3.465736            4.0       0.045955
4199     100.0  0.0      99.0   13.0 -1.856298            4.0       0.045955
5586     100.0  1.0      99.0    0.0 -1.519826            4.0       0.090465
5587     100.0  1.0      99.0    1.0 -3.465736            4.0       0.090465
5588     100.0  1.0      99.0    2.0 -3.465736            4.0       0.090465
5589     100.0  1.0      99.0    3.0 -1.268511            4.0       0.090465
5590     100.0  1.0      99.0    4.0 -3.465736            4.0       0.090465
5591     100.0  1.0      99.0    5.0 -3.465736            4.0       0.090465
5592     100.0  1.0      99.0    6.0 -3.465736            4.0       0.090465
5593     100.0  1.0      99.0    7.0 -3.465736            4.0       0.090465
5594     100.0  1.0      99.0    8.0 -3.465736            4.0       0.090465
5595     100.0  1.0      99.0    9.0 -3.465736            4.0       0.090465
5596     100.0  1.0      99.0   10.0 -3.465736            4.0       0.090465
5597     100.0  1.0      99.0   11.0 -3.465736            4.0       0.090465
```

5598	100.0	1.0	99.0	12.0	-3.465736	4.0	0.090465
5599	100.0	1.0	99.0	13.0	-1.856298	4.0	0.090465

	meaning_entropy
1386	0.742488
1387	0.742488
1388	0.742488
1389	0.650022
1390	0.742488
1391	0.650022
1392	0.742488
1393	0.629249
1394	0.650022
1395	0.742488
1396	0.742488
1397	0.650022
1398	0.629249
1399	0.742488
2786	0.485461
2787	0.485461
2788	0.485461
2789	0.522559
2790	0.485461
2791	0.522559
2792	0.485461
2793	0.485461
2794	0.522559
2795	0.485461
2796	0.485461
2797	0.522559
2798	0.503258
2799	0.503258
4186	0.000000
4187	0.000000
4188	0.000000
4189	0.000000
4190	0.081462
4191	0.000000
4192	0.162775
4193	0.000000
4194	0.000000
4195	0.082143
4196	0.000000
4197	0.081462
4198	0.000000
4199	0.082143
5586	0.162775
5587	0.000000
5588	0.000000
5589	0.223736
5590	0.000000
5591	0.081462
5592	0.000000
5593	0.082143
5594	0.162775
5595	0.082837
5596	0.000000

```
5597         0.084262
5598         0.082143
5599         0.081462
```

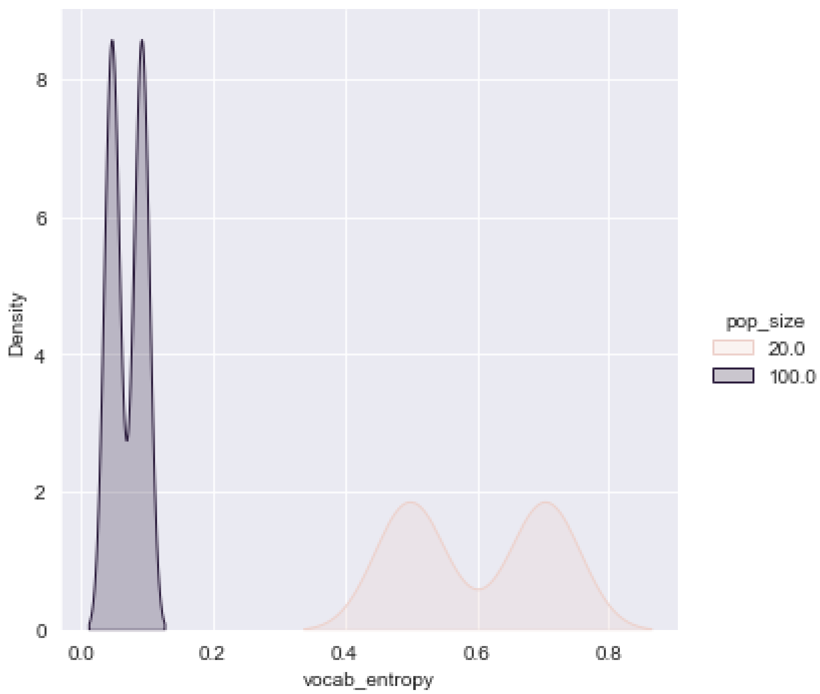
Just resetting the `t_timesteps` parameter to its original value, before we move on:

In [ ]:
```
t_timesteps = 5000
```

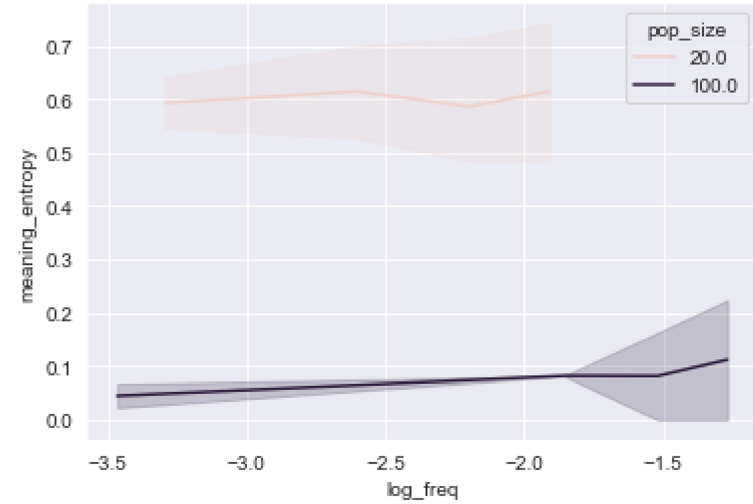
Plotting the results:

Now that we have a dataframe containing simulation results two different population sizes, we can plot the results using the three plotting functions defined above, as follows:

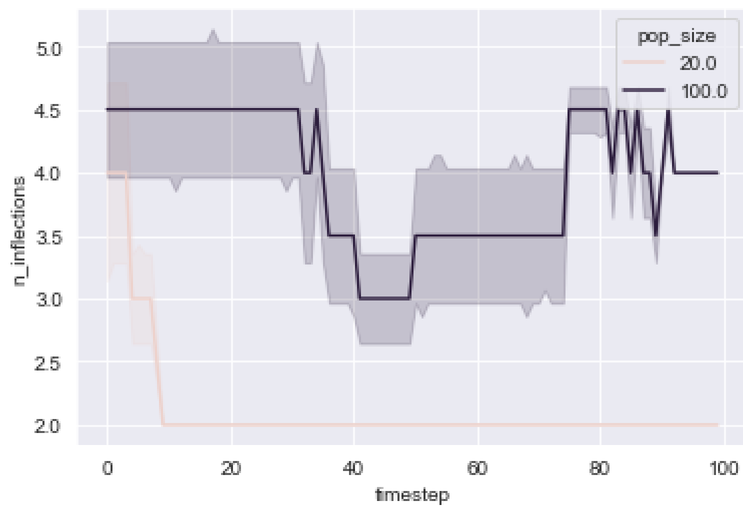
In [ ]:
```
%matplotlib inline

plot_vocab_entropy(final_timestep_results)
```



In [ ]:
```
%matplotlib inline

plot_meaning_entropy_by_freq(final_timestep_results)
```



In [ ]:
```
%matplotlib inline

plot_active_inflections_over_time(results_dataframe)
```



Exercise 4:

For this exercise, we need results of a simulation that is similar to Cuskley et al.'s "turnover" condition. Do you need to make any changes to the `replacement` parameter or `growth` parameter at the top of the notebook in order to do that? If not, feel free to re-use the `combined_dataframe` generated above.

Compare your own "vocab_entropy" plot with Fig. 2A of Cuskley et al. (2018).

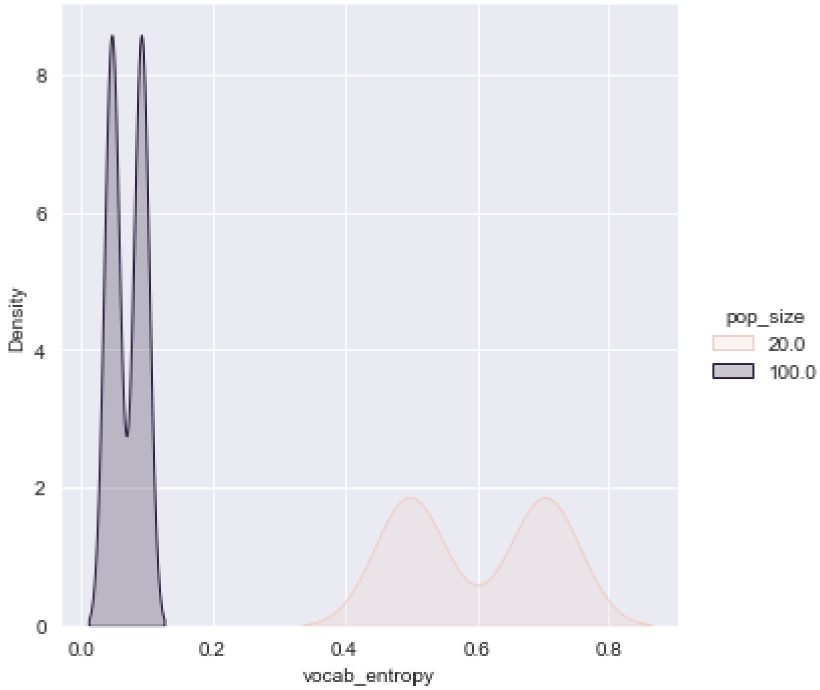
a) What does the measure H_v or "vocab_entropy" represent?

b) What does the measure H_l or "meaning_entropy" represent? And what does it mean to have an H_l of 0.0?

c) Does your own "vocab_entropy" plot look similar to Cuskley et al.'s Fig. 2A? Describe both the meaningful similarities (if any) and differences (if any). If you find meaningful differences, try to explain what the cause of those might be (based on what you know about the different parameter settings used here compared to Cuskley et al., 2018).

In [ ]:

```
%matplotlib inline

plot_vocab_entropy(final_timestep_results)
```



a) In this model, H_v or "vocab_entropy" represents the irregularity of inflections across lemmas: the higher the vocabulary entrophy is, the less regular the inflection system is.
b) H_l or "meaning_entropy" represents the irregularity of inflections for a particular lemma.

c) Overall distribution of vocab_entrophy for the two population sizes is simillar for the two figures: a higher population size correlates with a more regular inflection system. However, such pattern is more prominant in the above figure compared to Cuskley et al.'s Fig. 2A. This could be because the number of inflections used for the present model is 6 while it was 12 in the paper, allowing more variety in the combination of lemmas and inflections. As the small population is more likely to allow irregular inflections, an increase in the number of inflections could have been more evident for the small population.

Exercise 5:

For this exercise, run a simulation that is similar to Cuskley et al.'s "growth" condition. Do you need to make any changes to the `replacement` parameter or `growth` parameter at the top of the notebook in order to do that?

a) Which subfigure in Cuskley et al. (2018) does your "active_inflections_over_time" plot correspond to?

b) For this particular model and these simulations, how would you go about deciding how many timesteps you should ideally look it? When is it ok to stop running a simulation?

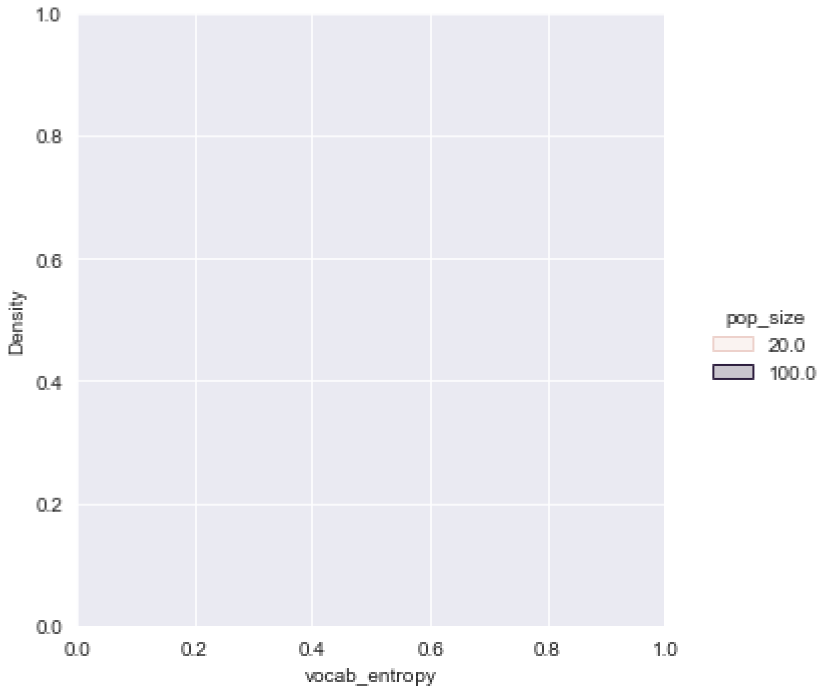
c) Looking at your own "active_inflections_over_time" plot, and comparing it to the corresponding subfigure in Cuskley et al. (2018), do you believe that you have run your simulations for enough timesteps? If not, explain why not. Or if you find that you cannot tell based on your plot, also explain why.

In [ ]:
```
t_timesteps = 500
replacement = False
growth = True

results_dataframe = run_multi_sizes(pop_sizes)
```

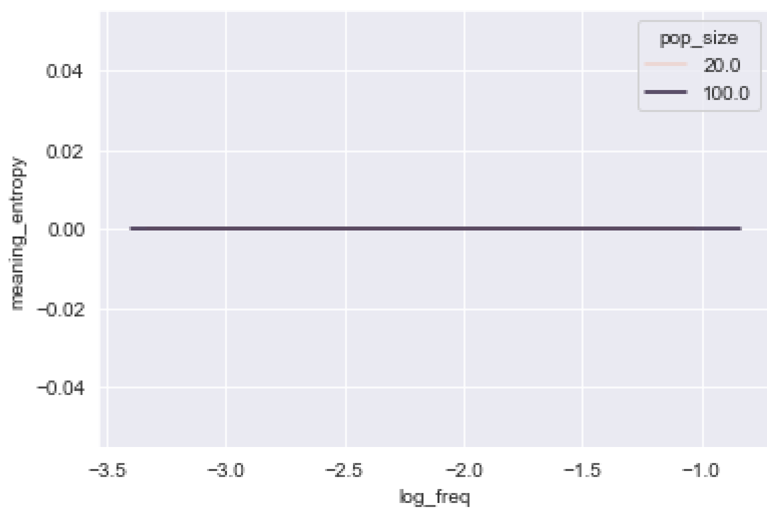
```
pop_size is:
20

r: 0
t: 0
self.running_popsize at end of simulation:
35

r: 1
t: 0
self.running_popsize at end of simulation:
46
Simulation(s) took 0.41 minutes to run

pop_size is:
100

r: 0
t: 0
self.running_popsize at end of simulation:
148

r: 1
t: 0
self.running_popsize at end of simulation:
195
Simulation(s) took 2.39 minutes to run
```

In [ ]:
```
%matplotlib inline
final_timestep_results = results_dataframe[results_dataframe["timestep"]==t_tir

plot_vocab_entropy(final_timestep_results)
```

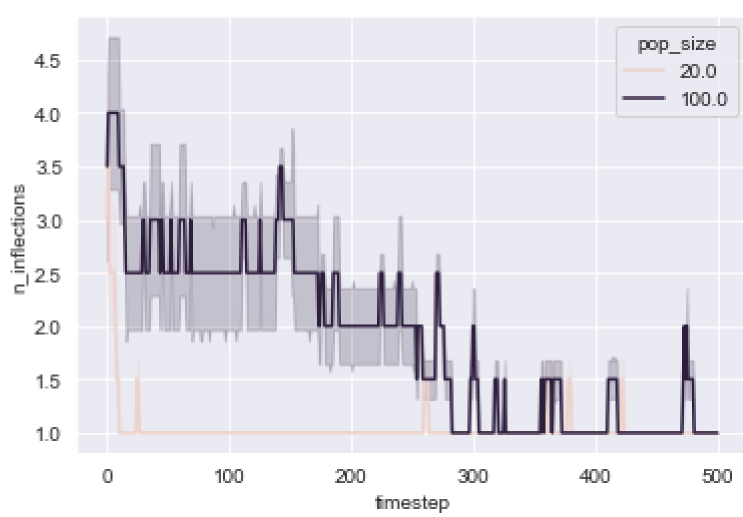
```
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages/
seaborn/distributions.py:316: UserWarning: Dataset has 0 variance; skipping de
nsity estimate. Pass `warn_singular=False` to disable this warning.
  warnings.warn(msg, UserWarning)
```



In [ ]:
```
plot_meaning_entropy_by_freq(final_timestep_results)
```



In [ ]:
```
plot_active_inflections_over_time(results_dataframe)
```



a) 3C
b) I believe one can end the simulation when the active inflection is one, which is the ultimate simplification of inflection system that is led by population turnover or growth.

c) I believe I have run enough simulations, as the active inflection is close to 0. However, Figure 3(C) in the paper shows the opposite pattern: the active inflection becomes larger over time. I don't know why this is the case, as addition of new learners should lead to simpler inflection system.

BONUS Exercise 6 (only if you have time left):

Read through the Simulation class more closely, and based on that, try to answer the following question:

Under which of the following conditions would you predict the run time of a simulation to increase most strongly?

1. If you double the number of lemmas in the vocabulary
2. If you double the number agents in the population
3. If you double the number of timesteps

In [ ]: