suspicion_calculations

August 16, 2025

1 Background

1.1 Setup

State space: integers from 0 to 20 sampled uniformly

Utterance space: all possible closed integer intervals within the state space that have a minimum range of 3 and maximum range of 5:

$$\mathcal{S} = \{0, 1, \dots, 20\}$$
$$\mathcal{U} = \{[0,2], [0,3], [0,4], \dots, [16,20], [17,20], [18,20]\}$$

1.2 Literal Listener: L_0

$$L_0(s \mid u) \propto [[u]](s)P(s)$$

$$[[u]](s) = \begin{cases} 1 & \text{if } a \le s \le b \\ 0 & \text{else} \end{cases} \quad \text{where } u = [a, b]$$

1.3 Pragmatic Speaker 1: S_1

We have α the rationality parameter, ψ the speaker type (inf, pers+, pers-), and β that switches between inf and pers (+ or -) (they do not use intermediate β values in the RSA opinion dynamics paper)

Persuasive strength does not depend on the state, and it gives more weight to utterances that have high or low expected value (relative to the literal listener) depending on pers+ or pers-.

I assign 0 probability for utterances with 0 informativity, so truth condition is satisfied for both values of β.

$$
\begin{aligned}
S_1(u|s,\psi) &\propto \mathrm{Inf}_{S_1}(u;s)^{\alpha\beta} \cdot \mathrm{PersStr}_{S_1}(u;\psi)^{\alpha(1-\beta)} \\
\mathrm{Inf}_{S_1}(u;s) &= L_0(s|u) \\
\mathrm{PersStr}_{S_1}(u;\psi) &= \begin{cases} \dfrac{E_{L_0}[s|u] - \min(S)}{\max(S) - \min(S)} & \text{if } \psi = \text{pers}^+ \\ \dfrac{\max(S) - E_{L_0}[s|u]}{\max(S) - \min(S)} & \text{if } \psi = \text{pers}^- \\ 1 & \text{if } \psi = \text{inf} \end{cases} \\
\beta &= \begin{cases} 1 & \text{if } \psi = \text{inf} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

1.4 Pragmatic Listener 1 (informative): L_1^{inf}

This is the standard pragmatic listener. Assumes the speaker is informative:

$$L_1^{\text{inf}}(s|u) \propto P(s) \cdot S_1(u|s, \text{"inf"})$$

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     import math
     import random
     import scipy.stats as stats

     def create_all_utterances(min_interval, max_interval, domain):
         min_val, max_val = domain
         domain_range = max_val - min_val + 1
         if min_interval < 1 or max_interval > domain_range or min_interval >=␣
      ↪max_interval:
             raise ValueError("Interval out of bounds of the domain.")
         utterances = []
         for interval_range in range(min_interval, max_interval + 1):
             for start in range(min_val, max_val + 1):
                 end = start + interval_range - 1
                 if end <= max_val:
                     utterances.append((start, end))

         return utterances

     def literal_listener(utterance, domain):
         min_val, max_val = domain
         start, end = utterance
         if start < min_val or end > max_val or start >= end:
             raise ValueError("Utterance out of bounds of the domain.")
```

```
    x = np.arange(min_val, max_val + 1)
    pmf = np.zeros(len(x))
    pmf[start: end + 1] = 1 / (end - start + 1)
    return (x, pmf)

def informativeness_all_utterances(state, utterances, domain):

    result = {}
    for utt in utterances:
        x, pmf = literal_listener(utt, domain)
        result[utt] = pmf[np.where(x == state)[0][0]]
    return result

def persuasiveness_all_utterances(pers, utterances, domain):

    result = {u: 0.0 for u in utterances}

    for utt in utterances:
        if pers == "inf":
            result[utt] = 1
        else:
            x, pmf = literal_listener(utt, domain)
            for i in range(len(pmf)):
                result[utt] += x[i] * pmf[i]
            result[utt] = (result[utt] - domain[0])/ (domain[1] - domain[0])
            if pers == "low":
                result[utt] = 1 - result[utt]
    return result

def pragmatic_speaker(state, pers, utterances, domain, alpha=1.0):
    # Compute informativeness and persuasiveness
    informativeness = informativeness_all_utterances(state, utterances, domain)
    persuasiveness = persuasiveness_all_utterances(pers, utterances, domain)

    if pers == "inf":
        beta = 1.0
    else:
        beta = 0.0
    # Compute softmax weights
    scores = []
    for utt in utterances:
        info = informativeness.get(utt, 0.0)
        pers_val = persuasiveness.get(utt, 0.0)

        if info > 0:
            score = (info ** (alpha * beta)) * (pers_val ** (alpha * (1 -␣
↪beta)))
```

```
        else:
            score = 0.0
        scores.append(score)

    scores = np.array(scores)
    probs = scores / np.sum(scores) if np.sum(scores) > 0 else np.
 ↪ones_like(scores) / len(scores)
    return {utt: p for utt, p in zip(utterances, probs)}

def pragmatic_listener_inf(utt, utterances, domain, alpha=3.0):
    state_prior = 1.0 / (domain[1] - domain[0] + 1)
    x = []
    pmf = []
    # inf case
    all_states = np.arange(domain[0], domain[1] + 1)
    for state in all_states:
        res_inf = pragmatic_speaker(state, "inf", utterances, domain,␣
 ↪alpha=alpha)[utt]
        x.append(state)
        pmf.append(res_inf * state_prior)
    pmf = np.array(pmf)
    pmf /= np.sum(pmf)
    return x, pmf
```

2 Calculating Suspicion

To calculate the suspicion, we first get the posterior probabilities for all utterances after the pragmatic listener hears an utterance u^*:

$$P(u'|u^*) = \sum_{s \in \mathcal{S}} S_1(u'|s, \text{"inf"}) \cdot L_1^{\text{inf}}(s|u^*)$$

We define an entropy/inverse-precision of an utterance as the entropy of the literal listener's state distribution after hearing that utterance, let's denote it as Ent(u):

$$\text{Ent(u)} = \sum_{s \in \mathcal{S}} \log_2(L_0(s|u)) \cdot L_0(s|u)$$

Using this posterior utterance probabilities, we get a distribution of entropy E, let's denote its PMF with $p_E(x)$.

$$p_E(x) = \sum_{u' \in \mathcal{U} \wedge \text{Ent}(u') = x} P(u'|u^*)$$

We want to calculate how surprising $\text{Ent}(u^*)$ considering the distribution $p_E(x)$, and use this surprisal as a quantity for suspicion towards persuasion. I may have misunderstood what we discussed as two different methods, but from what I understood, the first method is 1) the tail probability / p-value which is the method I understood firstly and found more intuitive.

1) tail probability / p-value is the probability that the speaker could have used an utterance with a lower entropy (higher precision) than u^*.

$$sus \propto \sum_{x<\mathrm{Ent}(u^*)} p_E(x)$$

I am not sure if I understood the second method correctly but to get a surprisal value we can use the 2) self-information / probability of $\mathrm{Ent}(u^*)$ given $p_E(x)$.

2) self-information / probability is the surprisal amount that $\mathrm{Ent}(u^*)$ carries:

$$sus \propto -\log_2(p_E(\mathrm{Ent}(u^*))) \quad \text{or} \quad sus \propto \frac{1}{p_E(\mathrm{Ent}(u^*))} \quad \text{or} \quad sus \propto 1 - p_E(\mathrm{Ent}(u^*))$$

Is this what we want? or something else?

[2]:
```
def get_entropy(utt, domain):
    x, pmf = literal_listener(utt, domain)
    pmf = pmf[pmf > 0]  # Filter out zero probabilities
    entropy = -np.sum(pmf * np.log2(pmf))
    return entropy

def posterior_utterance_distribution(utt, utterances, domain, alpha=3.0):
    utterance_probs = {u: 0.0 for u in utterances}
    all_states = np.arange(domain[0], domain[1] + 1)
    for state in all_states:
        x_state, pmf_state = pragmatic_listener_inf(utt, utterances, domain,␣
 ↪alpha=alpha)
        speaker_utt_probs = pragmatic_speaker(state, "inf", utterances, domain,␣
 ↪alpha=alpha)
        for u in utterances:
            utterance_probs[u] += pmf_state[x_state.index(state)] *␣
 ↪speaker_utt_probs[u]

    return utterance_probs

def entropy_distribution(utt, utterances, domain, alpha=3.0):
    utterance_probs = posterior_utterance_distribution(utt, utterances, domain,␣
 ↪alpha=alpha)
    entropies = {}
    for u, prob in utterance_probs.items():
        entropy = get_entropy(u, domain)
        entropies[entropy] = entropies.get(entropy, 0) + prob
    x, pmf = zip(*sorted(entropies.items()))
    return x, pmf
```

[4]:
```
def get_suspicion_pvalue(utt, utterances, domain, belief, alpha=3.0):
    x_ent, pmf_ent = entropy_distribution(utt, utterances, domain, alpha=alpha)
    utt_ent = get_entropy(utt, domain)
    suspicion = 0
    for i in range(len(x_ent)):
        if x_ent[i] < utt_ent:
            suspicion += pmf_ent[i]
    return suspicion
```