Speakers-and-Listeners

May 25, 2022

1 Preliminaries

[1]:

```
import json
import pandas as pd
import matplotlib.pylab as plt
import seaborn as sns
import itertools
import numpy as np
from scipy import stats

%load_ext autoreload
%autoreload 2
```

[2]:

```
from literal_listener import StatelessLiteralListener

from configuration import TRUE_REWARDS
```

[3]:

```
TEST_CONTEXT = [{'color': 'red', 'shape': 'circle'},
                {'color': 'red', 'shape': 'triangle'},
                {'color': 'blue', 'shape': 'square'}]
```

2 Literal Listener

2.1 Acting from Instructions

[4]:

```
listener = StatelessLiteralListener(alphaL=3)
```

[5]:

```
instruction_present = {"type": "instruction", "color": "red", "shape": "circle"}
print("\nUtterance: {}".format(instruction_present))
for a in TEST_CONTEXT:
    print("\t{} : {}".format(a, listener._prob_action_from_instruction(a,␣
 ↪TEST_CONTEXT, instruction_present)))

instruction_absent = {"type": "instruction", "color": "green", "shape":␣
 ↪"circle"}
print("\nUtterance: {}".format(instruction_absent))
for a in TEST_CONTEXT:
    print("\t{} : {}".format(a, listener._prob_action_from_instruction(a,␣
↪TEST_CONTEXT, instruction_absent)))
```

```
Utterance: {'type': 'instruction', 'color': 'red', 'shape': 'circle'}
        {'color': 'red', 'shape': 'circle'} : 1
        {'color': 'red', 'shape': 'triangle'} : 0
        {'color': 'blue', 'shape': 'square'} : 0

Utterance: {'type': 'instruction', 'color': 'green', 'shape': 'circle'}
        {'color': 'red', 'shape': 'circle'} : 0.3333333333333333
        {'color': 'red', 'shape': 'triangle'} : 0.3333333333333333
        {'color': 'blue', 'shape': 'square'} : 0.3333333333333333
```

2.2 Acting from Descriptions

Given a message u which consists of a feature-value tuple $(\phi, \mathbb{R})$, condition worlds w and return only those with are literally consistent with that utterance. Choose actions w.r.t. the posterior beliefs.

[6]:

```
description = {"type": "description", "feature": "red", "value":1}
print("Utterance: {}".format(description))
for a in TEST_CONTEXT:
    print("\t{} : {}".format(a, listener._prob_action_from_description(a,␣
↪TEST_CONTEXT, description)))


description_circle = {"type": "description", "feature": "circle", "value":1}
print("\nUtterance: {}".format(description_circle))
for a in TEST_CONTEXT:
    print("\t{} : {}".format(a, listener._prob_action_from_description(a,␣
↪TEST_CONTEXT, description_circle)))
```

```
Utterance: {'type': 'description', 'feature': 'red', 'value': 1}
        {'color': 'red', 'shape': 'circle'} : 0.4878555511603684
        {'color': 'red', 'shape': 'triangle'} : 0.4878555511603684
        {'color': 'blue', 'shape': 'square'} : 0.024288897679263205

Utterance: {'type': 'description', 'feature': 'circle', 'value': 1}
        {'color': 'red', 'shape': 'circle'} : 0.9094429985127419
        {'color': 'red', 'shape': 'triangle'} : 0.04527850074362907
        {'color': 'blue', 'shape': 'square'} : 0.04527850074362907
```

2.2.1 Present Rewards

[7]:

```
print("Instructions - Present Rewards")
print("\tGood Instruction: {}".format(listener.
 ↪present_rewards(instruction_present, TEST_CONTEXT, TRUE_REWARDS)))
print("\tNot Present Instruction: {}".format(listener.
 ↪present_rewards(instruction_absent, TEST_CONTEXT, TRUE_REWARDS)))

print("\nDescriptions - Present Rewards")
print("\tPresent Rewards: {}".format(listener.present_rewards(description,␣
 ↪TEST_CONTEXT, TRUE_REWARDS)))
print("\tPresent Rewards: {}".format(listener.
 ↪present_rewards(description_circle, TEST_CONTEXT, TRUE_REWARDS)))
```

```
Instructions - Present Rewards
        Good Instruction: 1
        Not Present Instruction: -0.6666666666666666

Descriptions - Present Rewards
        Present Rewards: 0.4149888581225788
        Present Rewards: 0.7736074962818548
```

2.2.2 Future Feature Counts / Rewards

[8]:

```
listener = StatelessLiteralListener(alphaL=3)
avg_features_advice = listener.future_feature_counts({"type": "description",␣
 ↪"feature": "green", "value": 2})

expected_rewards_advice = listener.feature_count_rewards(avg_features_advice,␣
 ↪TRUE_REWARDS)

string_feature_counts = ["{}:{:.2f}".format(k, v) for k, v in␣
 ↪avg_features_advice.items()]

print("Generalizability of \"Green is +2\" description:\n")
print("Future Features:")
for feature in string_feature_counts:
    print("\t{}".format(feature))
print("Future Rewards: {}".format(expected_rewards_advice))
```

```
Generalizability of "Green is +2" description:

Future Features:
        red:0.12
        square:0.33
        circle:0.33
        triangle:0.33
        green:0.76
        blue:0.12
Future Rewards: 1.2769904161327992
```

3 Literal Speaker

3.1 Utterance Preferences

3.1.1 Description-only feature shift

[9]:
```
from literal_speaker import LiteralSpeaker
from configuration import DESCRIPTIONS, ALL_UTTERANCES, EXP_UTTERANCES
```

[10]:
```
results_list = []
description_only_speaker = LiteralSpeaker(listener, alphaS=10,␣
 ↪utterances="descriptions")

for horizon in range(1, 11):

    probs = description_only_speaker.all_utterance_probabilities(TEST_CONTEXT,␣
 ↪horizon=horizon)

    prob_df = pd.DataFrame(DESCRIPTIONS)
    prob_df["prob"] = probs
    prob_df["horizon"] = horizon
    results_list.append(prob_df)

all_horizon_results = pd.concat(results_list)
```

3.1.2 Instruction to description shift

[11]:
```
import copy

results_list = []
utterances = ALL_UTTERANCES
all_utterances_speaker = LiteralSpeaker(listener, alphaS=10, utterances="all")

for horizon in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:

    probs = all_utterances_speaker.all_utterance_probabilities(TEST_CONTEXT,␣
 ↪horizon=horizon)

    prob_df = pd.DataFrame(utterances)
    prob_df["prob"] = probs
    prob_df["horizon"] = horizon
    results_list.append(prob_df)

all_horizon_results = pd.concat(results_list)
```

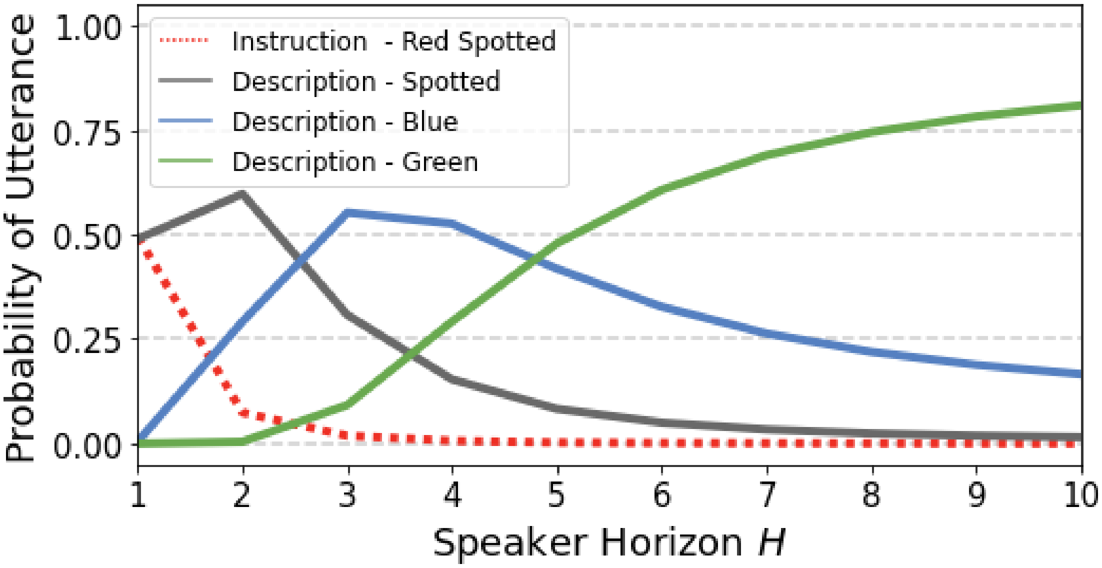
[12]:

```
green = all_horizon_results[all_horizon_results.feature == "green"]
blue = all_horizon_results[all_horizon_results.feature == "blue"]
circle = all_horizon_results[all_horizon_results.feature == "circle"]
instruction = all_horizon_results[(all_horizon_results.color == "red") &␣
 ↪(all_horizon_results["shape"] == 'circle')]

green_utt = "Description - Green"
blue_utt = "Description - Blue"
spotted_utt = "Description - Spotted"
instruct = "Instruction  - Red Spotted"

green["Utterance"] = green_utt
blue["Utterance"] = blue_utt
circle["Utterance"] = spotted_utt
instruction["Utterance"] = instruct

ordering = [instruct, spotted_utt, blue_utt, green_utt]

all_utts = pd.concat([green, blue, circle, instruction])

type_by_horizon = all_utts.groupby(["horizon", "Utterance"]).sum().
 ↪reset_index()[["horizon", "Utterance", "prob"]]

plt.figure(figsize=(8, 4))
ax = sns.lineplot(data=type_by_horizon, y='prob', x='horizon', linewidth=4,␣
 ↪hue='Utterance',
                  palette=['#ee2f24', 'dimgray', '#5580c1', '#69a94f'],␣
 ↪hue_order=ordering,
                  style='Utterance', dashes=['', '', '', (1,1)])

plt.ylabel("Probability of Utterance", size=18)
plt.legend(loc='best', fontsize=12)

for x in [0, .25, .5, .75, 1]:
    plt.axhline(x, alpha=.2, c='k', linestyle='--', zorder=0)

plt.yticks([0, .25, .5, .75, 1], fontsize=15);
plt.xticks(range(1, 11), fontsize=15);
plt.xlabel("Speaker Horizon $H$", fontsize=18)
plt.xlim(1, 10)
```

```
/var/folders/gv/42lb0z1j4dxf3wsk74nrxwx80000gn/T/ipykernel_60994/908865511.py:11
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
```

```
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  green["Utterance"] = green_utt
/var/folders/gv/421b0z1j4dxf3wsk74nrxwx80000gn/T/ipykernel_60994/908865511.py:12
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  blue["Utterance"] = blue_utt
/var/folders/gv/421b0z1j4dxf3wsk74nrxwx80000gn/T/ipykernel_60994/908865511.py:13
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  circle["Utterance"] = spotted_utt
/var/folders/gv/421b0z1j4dxf3wsk74nrxwx80000gn/T/ipykernel_60994/908865511.py:14
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  instruction["Utterance"] = instruct
```

```
[12]: (1.0, 10.0)
```



3.1.3 Multi-context: instruction-to-description

[13]:

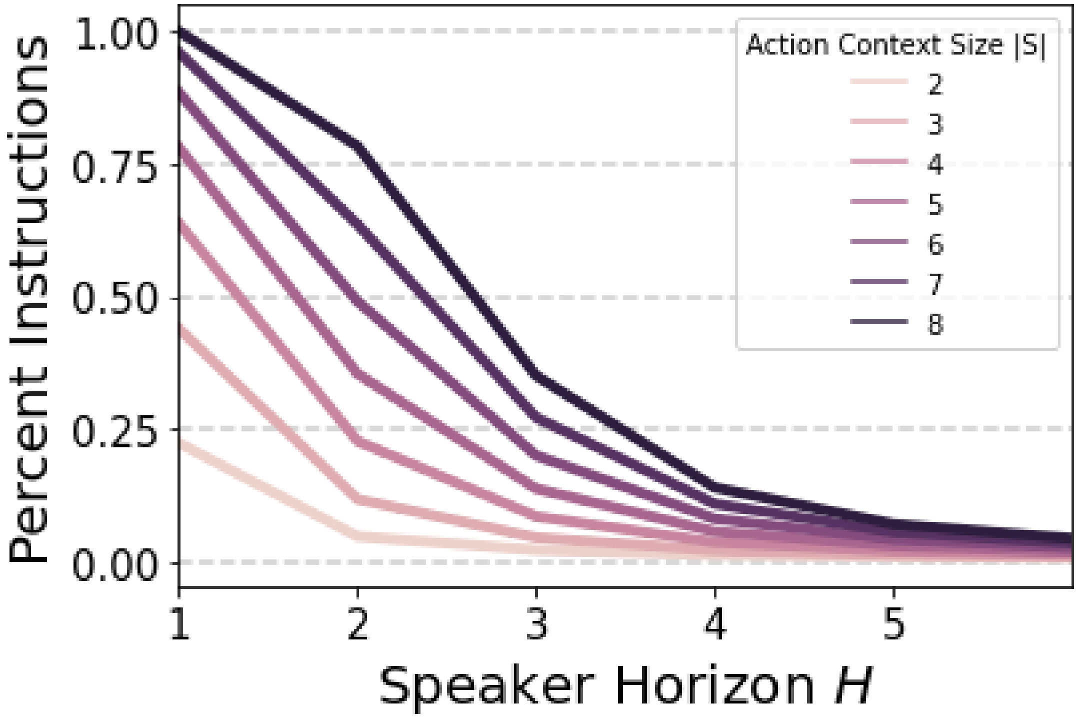
```
from configuration import UTTERANCES, ALL_STATES

from literal_listener import StatelessLiteralListener
from literal_speaker import LiteralSpeaker

def uttProbabilityS1(speaker_horizons, utterance_sets, contexts=ALL_STATES,␣
 ↪alphaS=10):

    results = []
    listener = StatelessLiteralListener()
    for utt_set in utterance_sets:

        for h in speaker_horizons:

            utt_probabilities = [0] * len(UTTERANCES[utt_set])
            speaker = LiteralSpeaker(listener=listener, utterances=utt_set,␣
 ↪alphaS=alphaS)

            for c in contexts:
                context_utt_probabilities = speaker.
 ↪all_utterance_probabilities(c, horizon=h)
                for i, u in enumerate(context_utt_probabilities):
                    utt_probabilities[i] += u

            df = pd.DataFrame(UTTERANCES[utt_set])
            df["horizon"] = h
            df["utterances"] = utt_set
            df["probs"] = utt_probabilities
            df["probs"] = df["probs"] / len(contexts)
            results.append(df)

    return pd.concat(results)
```

3.1.4 Increasing Context Size

[14]:

```
from configuration import ACTIONS

action_context_string = "Action Context Size |S|"

res = []
for size in range(2, 9):
    contexts = [list(l) for l in itertools.combinations(ACTIONS, size)]
    new_data = uttProbabilityS1(range(1, 11), ["all"], contexts=contexts)
    new_data[action_context_string] = size
    res.append(new_data)
```

```
df = pd.concat(res)

# type_by_horizon = df.groupby(["horizon", "Action Context Size", "type"]).
 ↪probs.sum().reset_index()
# sns.lineplot(data=type_by_horizon, x='horizon', y='probs', hue='Action␣
 ↪Context Size', style='type')

type_by_horizon = df[df["type"] == "instruction"].groupby(["horizon",␣
 ↪action_context_string]).probs.sum().reset_index()
sns.lineplot(data=type_by_horizon, x='horizon', y='probs',␣
 ↪hue=action_context_string, linewidth=4)

plt.xticks(range(0, 6), fontsize=15);
plt.ylabel("Percent Instructions", fontsize=20)
plt.xlabel("Speaker Horizon $H$", fontsize=20)
# plt.legend(fontsize=15)

plt.xlim(1, 6)

ys = [0, .25, .5, .75, 1.0]
for y in ys:
    plt.axhline(y, alpha=.2, c='k', linestyle='--', zorder=0)
plt.yticks(ys, fontsize=15);
```



3.2 Utterance Utility

[15]:
```
def literalUtilityFromS1(speaker_horizons, utterance_sets, contexts=ALL_STATES):

    results = []
    for utt_set in utterance_sets:

        for h in speaker_horizons:

            global_lit_rewards = 0
            local_lit_rewards = 0
            pct_instructions = 0
            pct_lies = 0

            speaker = LiteralSpeaker(listener=listener, utterances=utt_set)

            for c in contexts:
                utt_probabilities = speaker.all_utterance_probabilities(c,␣
↪horizon=h)

                for u, p in zip(UTTERANCES[utt_set], utt_probabilities):
                    if u["type"] == "instruction":
                        pct_instructions += p
                    if u["type"] == "description" and␣
↪TRUE_REWARDS[u["feature"]] != u["value"]:
                        pct_lies += p
                    global_lit_rewards += p * listener.future_rewards(u, None,␣
↪TRUE_REWARDS)
                    local_lit_rewards += p * listener.present_rewards(u, c,␣
↪TRUE_REWARDS)

            results.append({"global": global_lit_rewards/len(contexts),
                            "local": local_lit_rewards/len(contexts),
                            "pct_instructions": pct_instructions /␣
↪len(contexts),
                            "pct_lies": pct_lies / len(contexts),
                            "Utterance Set": utt_set,
                            "horizon": h})

    return pd.DataFrame(results)
```

[16]:
```
size_three_contexts = [list(l) for l in itertools.combinations(ACTIONS, 3)]
size_five_contexts = [list(l) for l in itertools.combinations(ACTIONS, 5)]

contexts_to_use = size_three_contexts
```

[17]:

```
res = literalUtilityFromS1(range(1, 11), ["instructions", "descriptions",␣
 ↪"all"], contexts=contexts_to_use)
res["objective_utility"] = ((res["horizon"] - 1) * res["global"] +␣
 ↪res["local"])/res["horizon"]
```

[18]:

```
rename_dict = {"instructions": "Instructions", "descriptions": "Descriptions",␣
 ↪"all": "All Utterances"}

res["Utterance Set"] = res["Utterance Set"].apply(lambda x: rename_dict[x])
```

[19]:

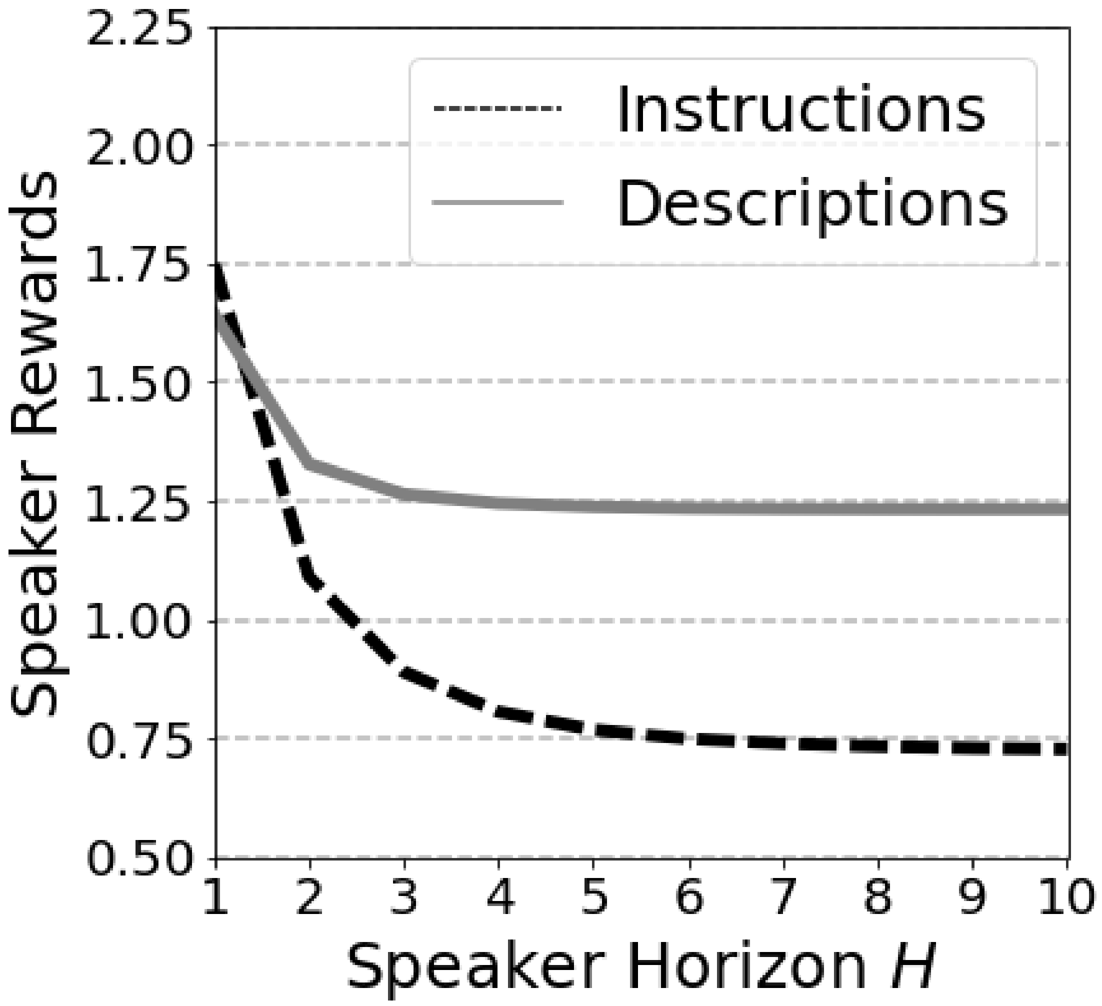
```
grilled_cheese = res.melt(id_vars=["Utterance Set", "horizon"],
                          value_vars=["global", "objective_utility", "local"],
                          var_name="Reward Type", value_name="rewards")
```

[20]:

```
to_plot = grilled_cheese[grilled_cheese["Utterance Set"] != "All Utterances"]

plt.figure(figsize=(6, 6))
g = sns.lineplot(data=to_plot[to_plot['Reward Type'] == "objective_utility"],␣
 ↪x='horizon', y='rewards', hue="Utterance Set",
                alpha=1,
                linewidth=5,
                hue_order=["Instructions", "Descriptions"],
                palette=["k", "gray"],
                style="Utterance Set", dashes=[(3,1), ''])

# plt.ylim(.4, 2)
ys = [.5, .75, 1, 1.25, 1.5, 1.75, 2, 2.25]

plt.yticks(ys, fontsize=20)
plt.xticks(fontsize=20)
# plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlim(1, 10)
for y in ys:
    plt.axhline(y, c='k', alpha=.3, linestyle='--', zorder=0)
plt.legend(loc='best', fontsize=25)
plt.ylabel("Speaker Rewards", fontsize=25)
plt.xlabel("Speaker Horizon $H$", fontsize=25)
```

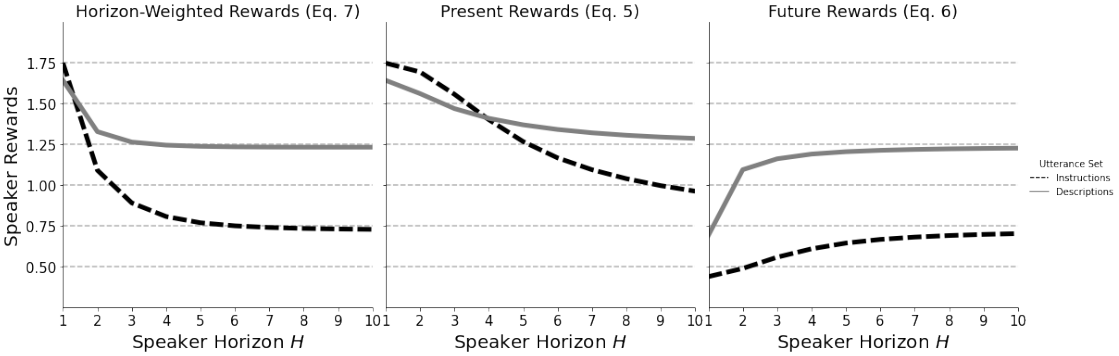
[20]:

```
Text(0.5, 0, 'Speaker Horizon $H$')
```



[21]:

```
to_plot = grilled_cheese[grilled_cheese["Utterance Set"] != "All Utterances"]

g = sns.relplot(data=to_plot, x='horizon', y='rewards', hue="Utterance Set",
                col='Reward Type', col_order=["objective_utility", "local",␣
 ↪"global"], kind='line',
                alpha=1,
                linewidth=5,
                hue_order=["Instructions", "Descriptions"],
                palette=["k", "gray"],
               style="Utterance Set", dashes=[(3,1), ''])


for i, ax in enumerate(g.axes.flat):  # set every-other axis for testing␣
 ↪purposes
    if i == 0:
        ax.set_ylabel("Speaker Rewards", fontsize=20)
        ax.set_title("Horizon-Weighted Rewards (Eq. 7)", fontsize=18)

    if i == 1:
        ax.set_title("Present Rewards (Eq. 5)", fontsize=18)

    if i == 2:
        ax.set_title("Future Rewards (Eq. 6)", fontsize=18)

    ax.set_xlabel("Speaker Horizon $H$", fontsize=20)
    ax.set_xticks(range(1,11))

    if contexts_to_use == size_three_contexts:
        ax.set_ylim(.25, 2)
        ys = [.5, .75, 1, 1.25, 1.5, 1.75]

    else:
        ax.set_ylim(.25, 2.5)
        ys = [.5, .75, 1, 1.25, 1.5, 1.75, 2, 2.25]

    ax.set_yticks(ys)
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax.set_xlim(1, 10)
    for y in ys:
        ax.axhline(y, c='k', alpha=.3, linestyle='--', zorder=0)
```

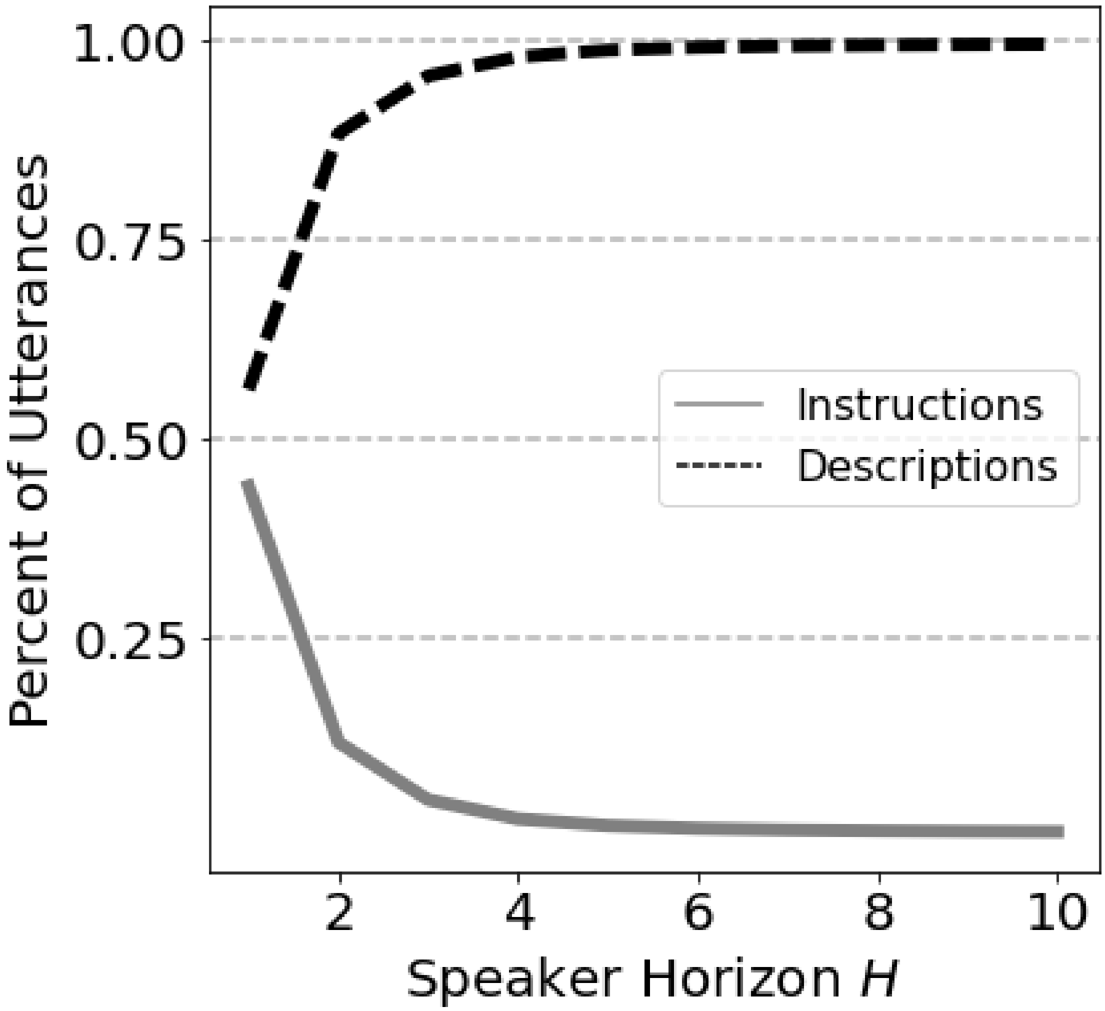


[22]:
```
all_utterances = res[res["Utterance Set"] == "All Utterances"]
all_utterances["pct_descriptions"] = 1 - all_utterances.pct_instructions

by_utt_type = all_utterances.melt(id_vars=["horizon"],
                                     value_vars=["pct_descriptions",␣
 ↪"pct_instructions"],
                                     var_name="Utterance Type",␣
 ↪value_name="Percent of Utterances")
```

```
rename_dict = {"pct_descriptions": "Descriptions", "pct_instructions":␣
 ↪"Instructions"}
by_utt_type["Utterance Type"] = by_utt_type["Utterance Type"].apply(lambda x:␣
 ↪rename_dict[x])

plt.figure(figsize=(6, 6))
g = sns.lineplot(data=by_utt_type, x='horizon', y='Percent of Utterances',␣
 ↪hue="Utterance Type",
                alpha=1,
                linewidth=5,
                hue_order=["Instructions", "Descriptions"],
                palette=["gray", "k"],
                style="Utterance Type", dashes=[(3,1), ''])

ys = [.25, .5, .75, 1]

plt.yticks(ys, fontsize=20)
plt.xticks(fontsize=20)

for y in ys:
    plt.axhline(y, c='k', alpha=.3, linestyle='--', zorder=0)
plt.legend(loc='best', fontsize=16)
plt.ylabel("Percent of Utterances", fontsize=20)
plt.xlabel("Speaker Horizon $H$", fontsize=20)
```

```
/var/folders/gv/42lb0z1j4dxf3wsk74nrxwx80000gn/T/ipykernel_60994/2820617515.py:2
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  all_utterances["pct_descriptions"] = 1 - all_utterances.pct_instructions
```

```
[22]: Text(0.5, 0, 'Speaker Horizon $H$')
```

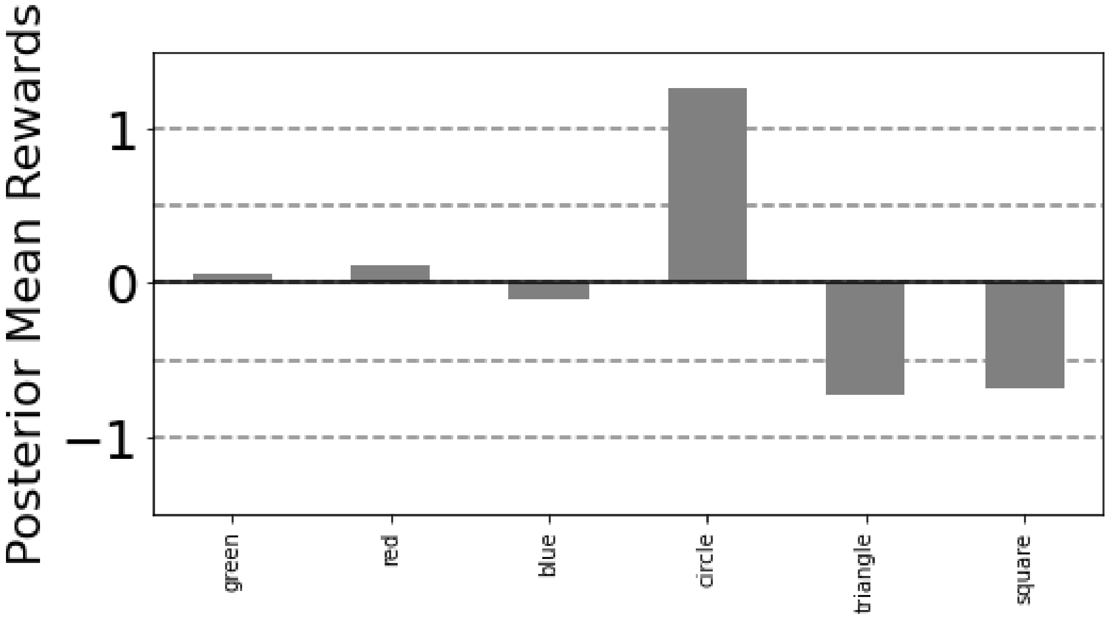


4 Pragmatic Listener

4.1 Beliefs from Utterances

```
[23]: from pragmatic_listener import PragmaticListener
      from visualizations import plot_point_estimate, plot_full_posterior,␣
       ↪plot_horizon_estimate

      base_listener = StatelessLiteralListener(alphaL=3)
      base_speaker = LiteralSpeaker(listener=base_listener, alphaS=10,␣
       ↪utterances="all")

      pragmatic_listener = PragmaticListener(base_speaker)
```

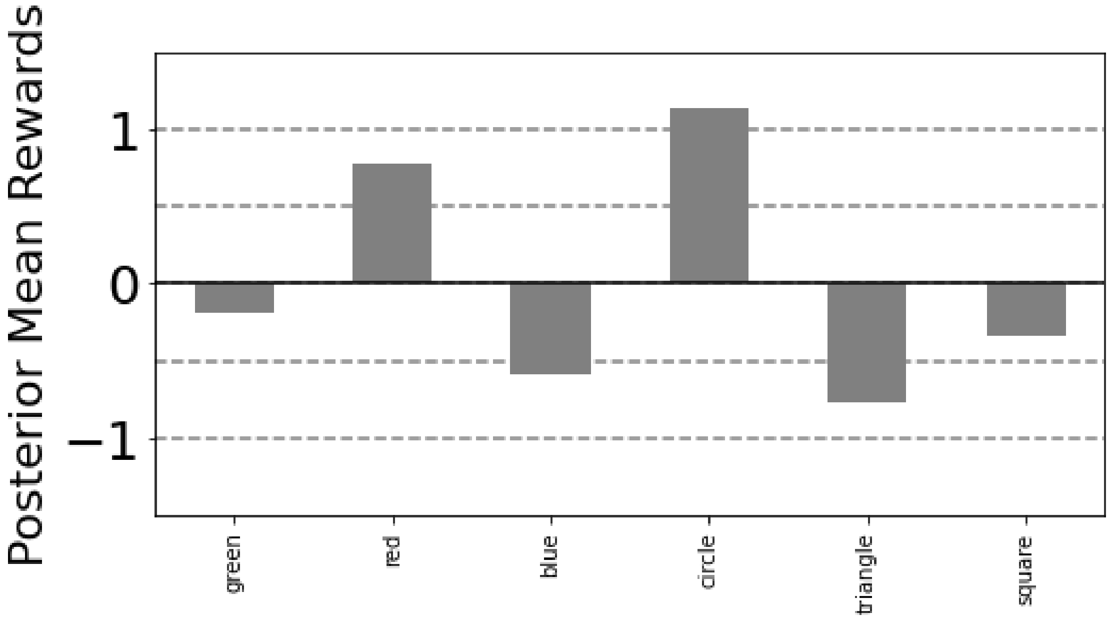
[24]:
```
description_to_use = {"type": "description", "feature": "circle", "value": 1}

description_posterior = pragmatic_listener.inference(description_to_use,␣
↪TEST_CONTEXT, horizon=[1, 2, 3, 4, 5, 10])
point_estimate = pragmatic_listener.
↪point_estimate_from_posterior(description_posterior)
plot_point_estimate(point_estimate)
```

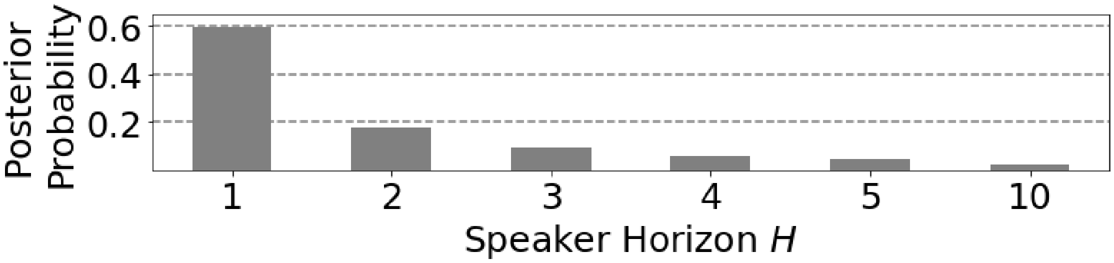


4.1.1 Inference over horizon

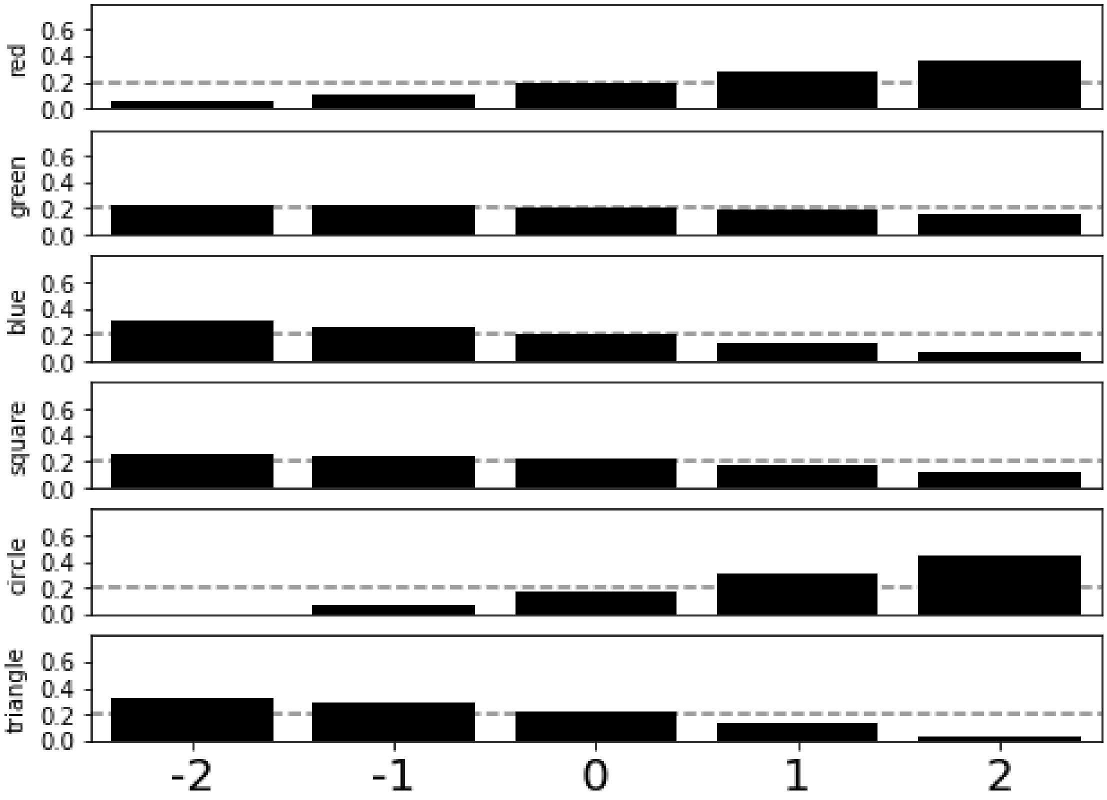
[25]:
```
from configuration import FEATURES

theory_to_mushrooms = {"red": "Red", "green": "Green", "blue": "Blue",
                       "circle": "Spotted", "triangle": "Solid", "square":␣
↪"Striped"}

MUSHROOM_FEATURES = [theory_to_mushrooms.get(k) for k in FEATURES]
```

[26]:
```
instruction = {"type": "instruction", "color": "red", "shape":"circle"}
reward_belief_df = pragmatic_listener.inference(instruction, TEST_CONTEXT,␣
↪horizon=[1, 2, 3, 4, 5, 10])

point_estimate = pragmatic_listener.
↪point_estimate_from_posterior(reward_belief_df)
plot_point_estimate(point_estimate)
plot_horizon_estimate(reward_belief_df)
```



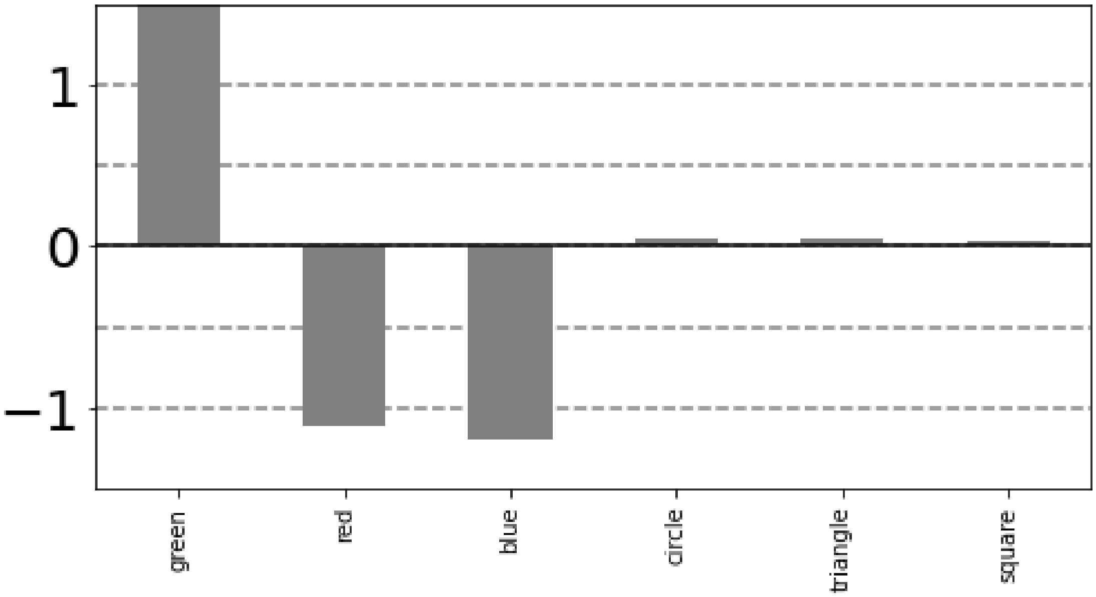


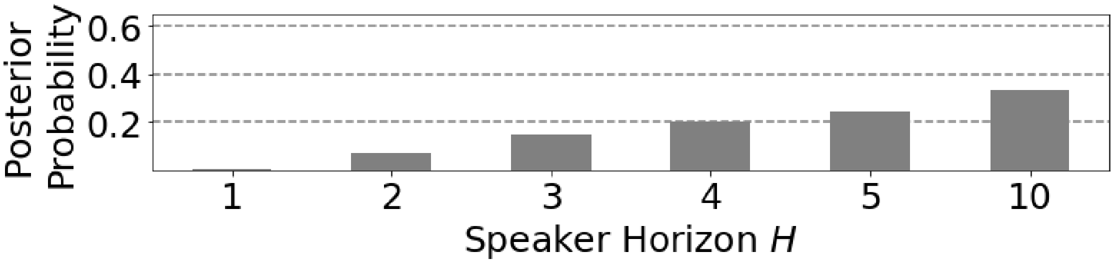
[27]:
```
plot_full_posterior(reward_belief_df, ylabel=True)
```



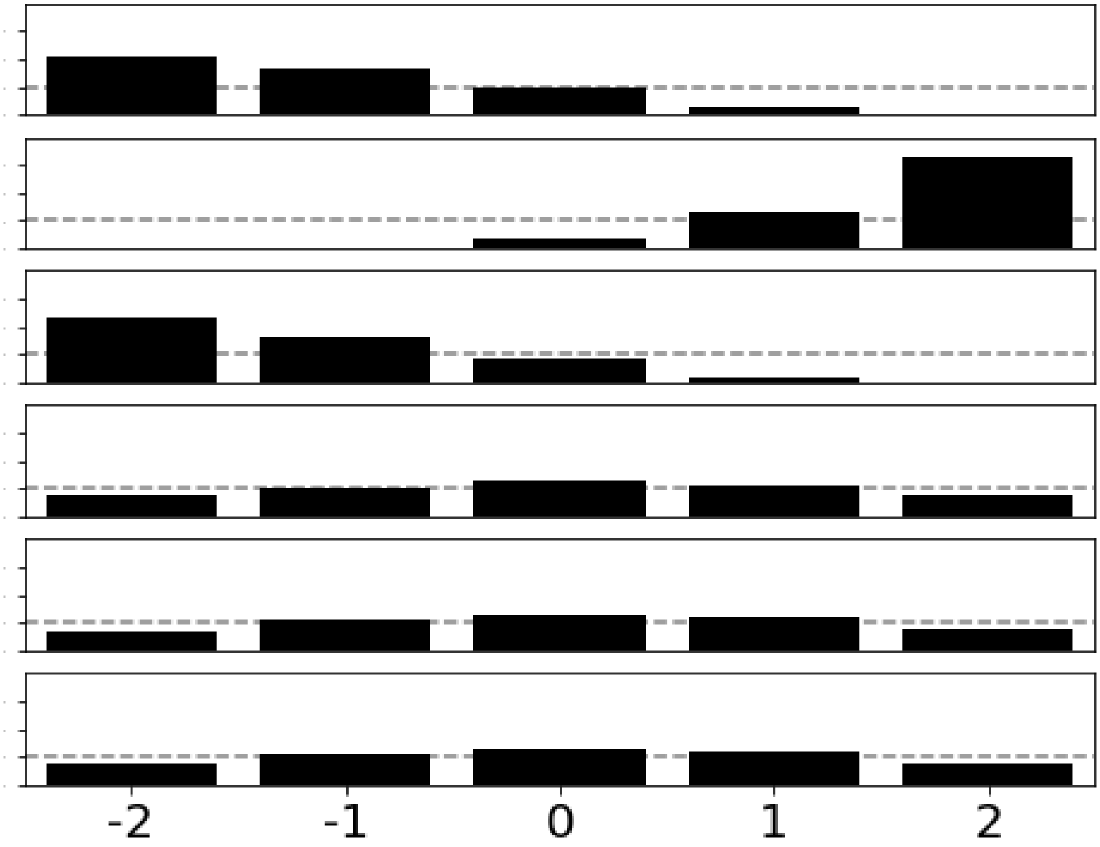
[28]:

```
description = {"type": "description", "feature": "green", "value":2}
reward_belief_df = pragmatic_listener.inference(description, TEST_CONTEXT,␣
↪horizon=[1, 2, 3, 4, 5, 10])

point_estimate = pragmatic_listener.
↪point_estimate_from_posterior(reward_belief_df)
plot_point_estimate(point_estimate, include_text=False)
plot_horizon_estimate(reward_belief_df, include_text=True, include_ticks=True)
```





[29]:
```
plot_full_posterior(reward_belief_df, ylabel=False)
```



4.2 Cache Pragmatic Inference

[30]:
```
import time
from os.path import exists

from configuration import ALL_STATES, UTTERANCES


##### Experimental settings #####
utterances_to_cache = "exp"
horizons = [1, 2, 4]
alphaSes = [3]

##### Theoretical settings ######
# utterances_to_cache = "all"
# horizons = [1, 2, 3, 4, 5, 10]
# alphaSes = [10]

for alphaS in alphaSes:
    to_cache_listener = StatelessLiteralListener()
    to_cache_speaker = LiteralSpeaker(listener=to_cache_listener,␣
↪utterances=utterances_to_cache, alphaS=alphaS)
    to_cache_pragmatic_listener = PragmaticListener(speaker=to_cache_speaker)

    for h in horizons:

        n_completed = 0
        n_in_horizon = len(ALL_STATES) * len(UTTERANCES[utterances_to_cache])
        print(f'Horizon {h}: {n_in_horizon} utterance-context pairs.')
        for c in ALL_STATES:
            for u in UTTERANCES[utterances_to_cache]:

                start_time_ms = round(time.time() * 1000)
                to_cache_pragmatic_listener.inference(u, c, h)
                end_time_ms = round(time.time() * 1000)

                n_seconds = (end_time_ms - start_time_ms)/1000
                if n_completed % 10 == 0 and n_seconds > 1:
                    print("\tRan #{} in {:.2f} seconds.".format(n_completed,␣
↪n_seconds))
                n_completed += 1
```

```
Horizon 1: 2100 utterance-context pairs.
Horizon 2: 2100 utterance-context pairs.
Horizon 4: 2100 utterance-context pairs.
```

5 Simulations for Paper

[31]:
```
def futureRewardsLiteralPragmatic(pragmatic_listener, speaker_horizon,␣
↪listener_horizons, utterance_set_name, contexts):
    """Given speaker / listener horizon(s) and utterances, return L0/L1 rewards.
↪"""

    pragmatic_rewards = 0
    literal_rewards = 0

    literal_listener = StatelessLiteralListener()

    for i, c in enumerate(contexts):

        utt_set = UTTERANCES[utterance_set_name]
        utt_probabilities = pragmatic_listener.speaker.
↪all_utterance_probabilities(c, horizon=speaker_horizon)

        for u, p in zip(utt_set, utt_probabilities):
            pragmatic_rewards += p * pragmatic_listener.future_rewards(u, c,␣
↪TRUE_REWARDS, listener_horizons)
            literal_rewards += p * literal_listener.future_rewards(u,␣
↪context=None, rewards=TRUE_REWARDS)

    return literal_rewards / len(contexts), pragmatic_rewards / len(contexts)
```

5.1 Simulation of literal / pragmatic listeners

Config: Theoretical

- `all` utterances
- `alphaS` = 10
- `horizon` = [1-10]

[32]:

```
# utterances_for_plot = "all"
# horizons_for_plot = [1, 2, 3, 4, 5, 10]
# alphaS = 10
```

Config: Experimental

- `exp` utterances
- `alphaS` = 3
- `horizon` = [1,2,4]

[33]:

```
utterances_for_plot = "exp"
horizons_for_plot = [1, 2, 4]
alphaS = 3
```

Run various simulations

[34]:

```
listener = StatelessLiteralListener()
speaker = LiteralSpeaker(listener, utterances=utterances_for_plot,␣
↪alphaS=alphaS)
pragmatic_listener = PragmaticListener(speaker)
```

Known Horizon

[35]:

```
results = []
for h in horizons_for_plot:
    print("Running horizon {}.".format(h))
    literal, pragmatic = futureRewardsLiteralPragmatic(pragmatic_listener, h,␣
↪h, utterances_for_plot, contexts=ALL_STATES)
    results.append({"horizon":h,
                    "alpha": a,
                    "literal": literal,
                    "pragmatic": pragmatic,
                    "pragmatic_diff": pragmatic-literal})

aligned_df = pd.DataFrame(results)
```

```
Running horizon 1.
Running horizon 2.
Running horizon 4.
```

Pedagogic Assumption ($H = 4$)

[36]:
```
results = []
for h in horizons_for_plot:
    literal, pragmatic = futureRewardsLiteralPragmatic(pragmatic_listener, h,␣
 ↪max(horizons_for_plot), utterances_for_plot, contexts=ALL_STATES)
    results.append({"horizon":h,
                    "literal": literal,
                    "pragmatic": pragmatic,
                    "pragmatic_diff": pragmatic-literal})

pedagogic_assumption_df = pd.DataFrame(results)
```

Locally-optimal assumption ($H = 1$)

[37]:
```
results = []
for h in horizons_for_plot:
    literal, pragmatic = futureRewardsLiteralPragmatic(pragmatic_listener, h,␣
 ↪1, utterances_for_plot, contexts=ALL_STATES)
    results.append({"horizon":h,
                    "literal": literal,
                    "pragmatic": pragmatic,
                    "pragmatic_diff": pragmatic-literal})

conservative_listener = pd.DataFrame(results)
```

5.1.1 Uncertain Pragmatic Listener

[38]:
```
results = []
for h in horizons_for_plot:
    literal, pragmatic = futureRewardsLiteralPragmatic(pragmatic_listener, h,␣
 ↪horizons_for_plot, utterances_for_plot, contexts=ALL_STATES)
    results.append({"horizon":h,
                    "literal": literal,
                    "pragmatic": pragmatic,
                    "pragmatic_diff": pragmatic-literal})

joint_inference_df = pd.DataFrame(results)
```

[39]:

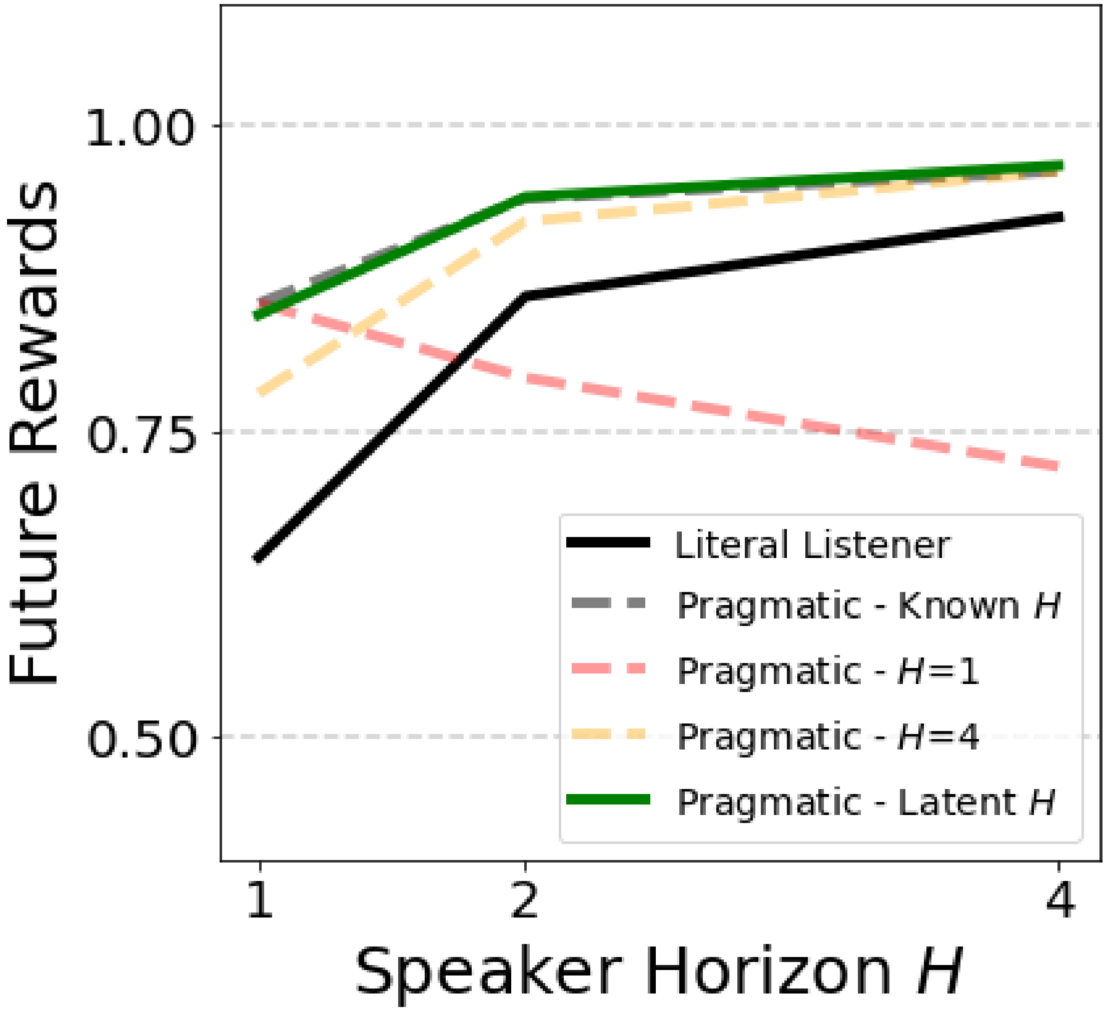
```
aligned_df["calibration"] = "Known Horizon"
pedagogic_assumption_df["calibration"] = "Pedagogic Assumption (H=4)"
conservative_listener["calibration"] = "Conservative Assumption (H=1)"
joint_inference_df["calibration"] = "Joint Inference"

full_df = pd.concat([aligned_df, pedagogic_assumption_df,␣
 ↪conservative_listener, joint_inference_df])
```

[40]:

```
full_df["horizon"] = full_df["horizon"].apply(lambda x: 7 if x == 10 else x)

plt.figure(figsize=(6,6))

known_horizon = full_df[full_df["calibration"] == "Known Horizon"]
plt.plot(known_horizon.horizon, known_horizon.literal, c='k', linewidth=4,␣
 ↪alpha=1, label="Literal Listener")

plt.plot(known_horizon.horizon, known_horizon.pragmatic, c='k', linestyle='--',␣
 ↪linewidth=4, alpha=.5, label="Pragmatic - Known $H$")

conservative = full_df[full_df["calibration"] == "Conservative Assumption␣
 ↪(H=1)"]
plt.plot(conservative.horizon, conservative.pragmatic, c='r', linestyle='--',␣
 ↪linewidth=4, alpha=.4, label="Pragmatic - $H$=1")

myopic_speaker = full_df[full_df["calibration"] == "Pedagogic Assumption (H=4)"]
plt.plot(myopic_speaker.horizon, myopic_speaker.pragmatic, c='orange',␣
 ↪linestyle='--', linewidth=4, alpha=.4, label=f'Pragmatic -␣
 ↪$H$={max(horizons_for_plot)}')


joint = full_df[full_df["calibration"] == "Joint Inference"]
plt.plot(joint.horizon, joint.pragmatic, c='g', linewidth=4, alpha=1,␣
 ↪label="Pragmatic - Latent $H$")


plt.ylabel("Future Rewards", size=25)
plt.xlabel("Speaker Horizon $H$", fontsize=25)

if horizons_for_plot == [1, 2, 3, 4, 5, 10]:

    plt.xticks([1, 2, 3, 4, 5, 7], labels=[1, 2, 3, 4, 5, 10], size=20);
    plt.ylim(.45, 1.30)
    yticks = [.5, .75, 1, 1.25]
    plt.yticks(yticks, size=20)

else:
    plt.xticks([1, 2, 4], size=20)
    plt.ylim(.4, 1.1)
    yticks = [.5, .75, 1]
    plt.yticks(yticks, size=20)

for y in yticks:
    plt.axhline(y, linestyle='--', c='k', alpha=.2, zorder=0)

# plt.show()

plt.legend(loc='best', fontsize=14)
```

[40]:

```
<matplotlib.legend.Legend at 0x7f977bd4b880>
```



[41]:

```
full_df.groupby(['calibration', 'horizon']).pragmatic.mean()
```

```
[41]: calibration                    horizon
      Conservative Assumption (H=1)  1          0.854550
                                     2          0.793714
                                     4          0.720938
      Joint Inference                1          0.845070
                                     2          0.941477
                                     4          0.967242
      Known Horizon                  1          0.854550
                                     2          0.940189
                                     4          0.962383
      Pedagogic Assumption (H=4)     1          0.780738
                                     2          0.921340
                                     4          0.962383
      Name: pragmatic, dtype: float64
```

6 Behavioral Data

```
[42]: human_utterances = json.load(open("data/exp_utterances.json"))
```

6.1 Analysis of chosen utterances

```
[43]: def literalUtilityFromUtterances(human_utterances, contexts=ALL_STATES):

          results = []
          horizons = list(set([u["horizon"] for u in human_utterances]))

          for h in horizons:

              h_utterances = [u for u in human_utterances if u["horizon"] == h]
              global_lit_rewards = 0
              local_lit_rewards = 0
              instructions = 0
              lies = 0

              utterances = [d["utt"] for d in h_utterances]
              contexts = [d["action_context"] for d in h_utterances]
              for u, c in zip(utterances, contexts):

                  if u["type"] == "instruction":
                      instructions += 1
                  if u["type"] == "description" and TRUE_REWARDS[u["feature"]] !=␣
        ↪u["value"]:
                      lies += 1

                  global_lit_rewards += listener.future_rewards(u, None, TRUE_REWARDS)
                  local_lit_rewards += listener.present_rewards(u, c, TRUE_REWARDS)
              results.append({"global": global_lit_rewards/len(h_utterances),
                              "local": local_lit_rewards/len(h_utterances),
                              "pct_instructions": instructions / len(h_utterances),
                              "pct_lies": lies / len(h_utterances),
                              "n_utterances": len(h_utterances),
                              "horizon": h})

          return pd.DataFrame(results)
```

[44]:

```
res = literalUtilityFromUtterances(human_utterances)
res["objective_utility"] = (res["local"] + (res["horizon"]-1) * res["global"])/
 ↪res["horizon"]
```

[45]:

```
grilled_cheese = res.melt(id_vars=["horizon"],
                          value_vars=["global", "objective_utility", "local"],
                          var_name="Reward Type", value_name="rewards")
```

6.2 Analysis of pragmatics

[46]:

```
def futureRewardsFromExperiment(pragmatic_listener, human_utterances,␣
 ↪horizons=None):
    """Given speaker / listener horizon(s) and utterances, return L0/L1 rewards.
 ↪"""

    results = []
    literal_listener = StatelessLiteralListener()

    if horizons is None:
        horizons = list(set([u["horizon"] for u in human_utterances]))

    for h in horizons:

        h_utterances = [u for u in human_utterances if u["horizon"] == h]

        print(f"Horizon {h}: {len(h_utterances)} utterances.")

        for i, u in enumerate(h_utterances):

            u = copy.deepcopy(u)

            literal = literal_listener.future_rewards(u["utt"], context=None,␣
 ↪rewards=TRUE_REWARDS)

            pragmatic_aligned = pragmatic_listener.future_rewards(u["utt"],␣
 ↪u["action_context"], TRUE_REWARDS, h)
```

```
            pragmatic_conservative = pragmatic_listener.
↪future_rewards(u["utt"], u["action_context"], TRUE_REWARDS, 1)
            pragmatic_long_horizon = pragmatic_listener.
↪future_rewards(u["utt"], u["action_context"], TRUE_REWARDS, 4)
            pragmatic_uncertain = pragmatic_listener.future_rewards(u["utt"],␣
↪u["action_context"], TRUE_REWARDS, horizons)

            uncertain_posterior = pragmatic_listener.inference(u["utt"],␣
↪u["action_context"], horizons)
            horizon_estimate = uncertain_posterior.
↪multiply(uncertain_posterior["probability"], axis='index').apply(np.
↪sum)["horizon"]
            point_estimate = pragmatic_listener.
↪point_estimate_from_posterior(uncertain_posterior)

            u["literal"] = literal
            u["pragmatic_aligned"] = pragmatic_aligned
            u["pragmatic_uncertain"] = pragmatic_uncertain

            u["pragmatic_conservative"] = pragmatic_conservative
            u["pragmatic_pedagogic"] = pragmatic_long_horizon

            u["horizon_estimate"] = horizon_estimate
            u["point_estimate"] = point_estimate

            results.append(u)

    return results
```

[47]:

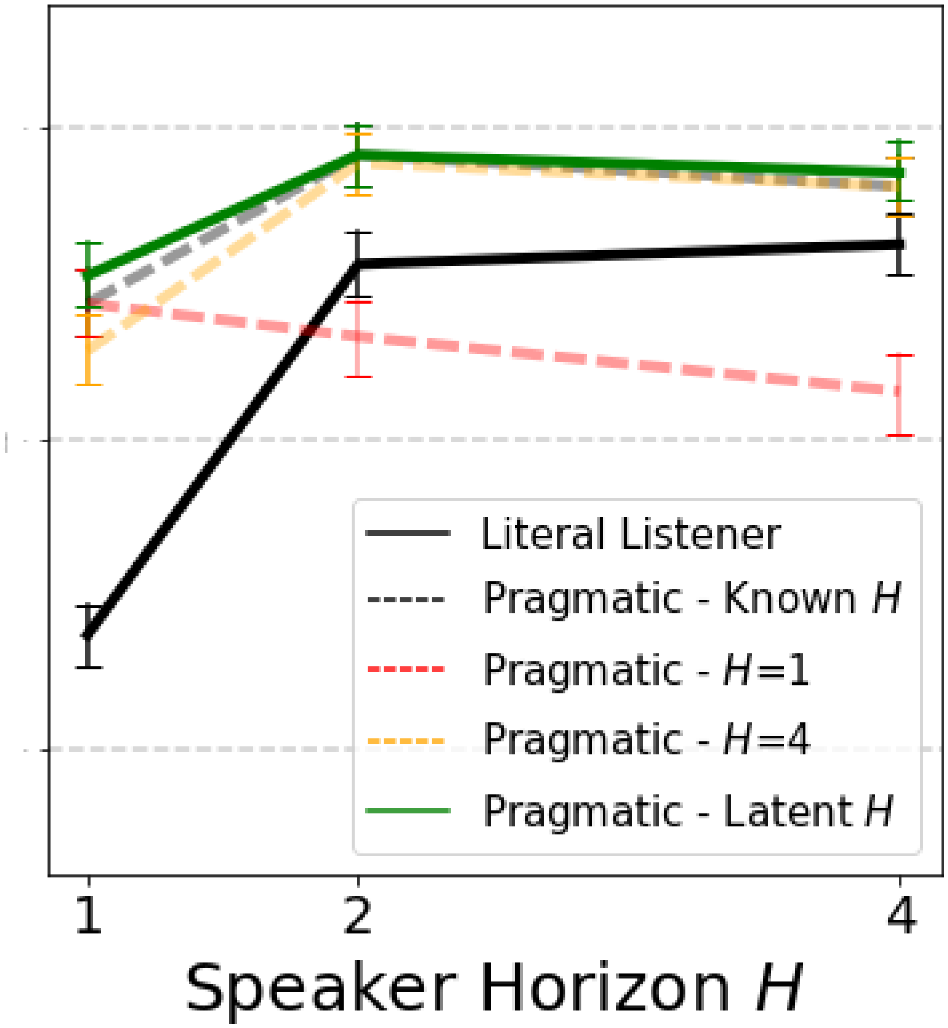
```
alphaS = 3

all_results = []

literal = StatelessLiteralListener()
speaker = LiteralSpeaker(literal, utterances="exp", alphaS=alphaS)
pragmatic_listener = PragmaticListener(speaker)

results = futureRewardsFromExperiment(pragmatic_listener, human_utterances)

res = pd.DataFrame(results)
res["alphaS"] = alphaS
all_results.append(res)

res = pd.concat(all_results)
```

```
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
```

[48]:
```
rename_dict = {"pragmatic_aligned": "Pragmatic - Known $H$",
               "pragmatic_uncertain": "Pragmatic - Latent $H$",
               "pragmatic_conservative": "Pragmatic - $H$=1",
               "pragmatic_pedagogic": "Pragmatic - $H$=4",
               "literal": "Literal Listener"}

to_plot_human_data = res.melt(id_vars="horizon", var_name="listener",␣
 ↪value_name="rewards",
                   value_vars=["literal", "pragmatic_aligned",␣
 ↪"pragmatic_uncertain", "pragmatic_conservative", "pragmatic_pedagogic"])

to_plot_human_data["listener"] = to_plot_human_data.listener.apply(lambda x:␣
 ↪rename_dict[x])
```

[49]:
```
plt.figure(figsize=(6,6))

sns.lineplot(data=to_plot_human_data, x='horizon', y="rewards", hue="listener",
             ci=95, err_style='bars', err_kws={"capsize": 5},
             hue_order=["Literal Listener", "Pragmatic - Known $H$", "Pragmatic␣
 ↪- $H$=1", "Pragmatic - $H$=4", "Pragmatic - Latent $H$"],
             palette=["k", "k", "r", 'orange', "g"],  linewidth=4, alpha=1,␣
 ↪style='listener', dashes=['', (3,1), '', (3,1), (3,1)])

plt.xticks([1, 2, 4], size=20)
plt.xlabel("Speaker Horizon $H$", fontsize=25)

yticks = [.5, .75, 1, 1.25]
plt.yticks(yticks, size=0)
for y in yticks:
    plt.axhline(y, linestyle='--', c='k', alpha=.2, zorder=0)

plt.ylabel("Future Rewards", fontsize=0)

ax_children = plt.gca().get_children()
print()

# pragmatic - known h
plt.setp([ax_children[9]],alpha=.4)

# pragmatic- h = 1
plt.setp([ax_children[2]],alpha=.4)
plt.setp([ax_children[13]],alpha=.4)

# pragmatic- h = 4
plt.setp([ax_children[17]],alpha=.4)
plt.legend(fontsize=15)
plt.ylim(.4, 1.1)
```

```
[49]: (0.4, 1.1)
```



```
[50]: from configuration import utt_to_string, context_to_string

res["item_key"] = res.apply(lambda x:␣
 ↪f'{utt_to_string(x["utt"])}|{context_to_string(x["action_context"])}|{x["workerid"]}',␣
 ↪axis=1)
```

6.2.1 Calculate means / export to R

```
[51]: res_to_export = res.drop(['action_context', 'workerid', 'horizon', 'utt',␣
       ↪'horizon_estimate', 'point_estimate', 'alphaS', 'item_key"], axis=1)
      res_to_export.to_csv("utterance_posterior_rewards.csv", index=False)
```

```
[52]: res[['literal', 'pragmatic_uncertain', 'pragmatic_aligned',␣
       ↪'pragmatic_conservative', 'pragmatic_pedagogic']].describe().round(2)
```

```
[52]:        literal  pragmatic_uncertain  pragmatic_aligned  \
      count  2772.00              2772.00            2772.00
      mean      0.79                 0.94               0.93
      std       0.41                 0.39               0.40
      min      -0.75                -1.34              -1.34
      25%       0.50                 0.64               0.65
      50%       0.75                 1.02               1.03
      75%       1.28                 1.25               1.26
      max       1.28                 1.58               1.60

             pragmatic_conservative  pragmatic_pedagogic
      count                 2772.00              2772.00
      mean                     0.83                 0.91
      std                      0.46                 0.40
      min                     -1.36                -1.32
      25%                      0.47                 0.63
      50%                      0.90                 1.05
      75%                      1.20                 1.25
      max                      1.60                 1.48
```

6.3 Appendix E: Pragmatic Inference Details

```
[53]: res["instruct_color"] = res.utt.apply(lambda x: x.get("color"))
      res["instruct_shape"] = res.utt.apply(lambda x: x.get("shape"))
      res["descript_feature"] = res.utt.apply(lambda x: x.get("feature",␣
       ↪"instruction"))
      res["descript_value"] = res.utt.apply(lambda x: x.get("value"))
      res["instruction"] = res.utt.apply(lambda x: x.get("type") == "instruction")
```

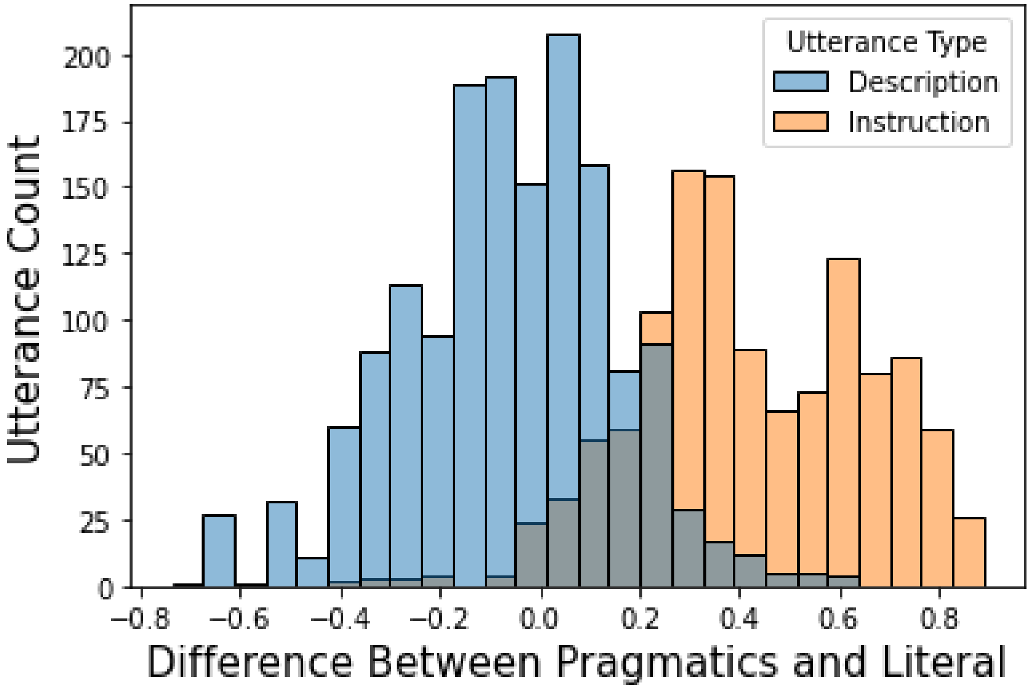
```
[54]: res["pragmatic_aligned_diff"] = res.pragmatic_aligned - res.literal
      res["pragmatic_uncertain_diff"] = res.pragmatic_uncertain - res.literal
```

```
[55]: res["Utterance Type"] = res.instruction.apply(lambda x: "Instruction" if x else␣
       ↪"Description")
```

```
[56]: sns.histplot(data=res, hue="Utterance Type", x='pragmatic_uncertain_diff')
      plt.xlabel("Difference Between Pragmatics and Literal", fontsize=15)
      plt.ylabel("Utterance Count", fontsize=15)
```

[56]:

```
Text(0, 0.5, 'Utterance Count')
```



[57]:

```
res.groupby("Utterance Type")[["pragmatic_uncertain_diff"]].describe().round(3)
```

[57]:

```
               pragmatic_uncertain_diff                                          \
                                  count   mean    std    min    25%    50%
Utterance Type
Description                      1569.0 -0.067  0.215 -0.739 -0.190 -0.052
Instruction                      1203.0  0.423  0.232 -0.594  0.268  0.395



                  75%    max
Utterance Type
Description     0.076  0.617
Instruction     0.616  0.890
```

[58]:

```
pragmatic_difference_descriptions = res[~res.instruction].
 ↪pragmatic_uncertain_diff
stats.ttest_1samp(pragmatic_difference_descriptions, 0)
```

[58]:

```
Ttest_1sampResult(statistic=-12.288703528317031, pvalue=3.32739264533416e-33)
```

[59]:

```
pragmatic_difference_instructions = res[res.instruction].
 ↪pragmatic_uncertain_diff
```

```
stats.ttest_1samp(pragmatic_difference_instructions, 0)
```

```
[59]: Ttest_1sampResult(statistic=63.34314897797516, pvalue=0.0)
```

```
[60]: res.groupby("descript_feature").pragmatic_uncertain_diff.describe().round(3)
```

```
[60]:                     count   mean    std    min    25%    50%    75%    max
      descript_feature
      blue                389.0 -0.196  0.271 -0.644 -0.353 -0.236 -0.016  0.520
      circle              213.0  0.083  0.147 -0.360 -0.003  0.070  0.188  0.468
      green               860.0 -0.050  0.156 -0.402 -0.151 -0.049  0.050  0.378
      instruction        1203.0  0.423  0.232 -0.594  0.268  0.395  0.616  0.890
      square              107.0 -0.023  0.246 -0.739 -0.134 -0.043  0.076  0.617
```

```
[61]: res.groupby(["instruct_color", "instruct_shape"]).pragmatic_aligned_diff.
       ↪describe()
```

```
[61]:                                  count      mean       std       min       25%  \
      instruct_color instruct_shape
      blue           circle           16.0  0.464016  0.405353 -0.239676  0.109121
                     square            2.0 -0.519293  0.100403 -0.590288 -0.554790
                     triangle          4.0 -0.283082  0.171797 -0.526259 -0.337772
      green          circle          514.0  0.470636  0.275594 -1.213894  0.377254
                     square          152.0  0.431725  0.309199 -0.318704  0.200191
                     triangle        266.0  0.388986  0.280168 -0.500000  0.152600
      red            circle          142.0  0.397844  0.168106 -0.045301  0.275930
                     square           24.0  0.324693  0.194851  0.014863  0.147987
                     triangle         83.0  0.326271  0.177794 -0.235731  0.205801

                                          50%       75%       max
      instruct_color instruct_shape
      blue           circle          0.557623  0.886528  0.886528
                     square         -0.519293 -0.483795 -0.448297
                     triangle       -0.234915 -0.180225 -0.136238
      green          circle          0.492556  0.639280  0.851747
                     square          0.385381  0.645880  1.023056
                     triangle        0.337304  0.586262  0.948830
      red            circle          0.397385  0.481486  0.687240
                     square          0.326866  0.541763  0.541763
                     triangle        0.303651  0.407212  0.625669
```

6.4 Appendix C: Choosing β_{S_1}

```
[62]: alphaTestResults = []
      horizons = [1, 2, 4]
      alphas = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
for alpha in alphas:

    print(f'AlphaS: {alpha}.')

    literal = StatelessLiteralListener()
    speaker = LiteralSpeaker(literal, utterances="exp", alphaS=alpha)
    pragmatic_listener = PragmaticListener(speaker)

    results = futureRewardsFromExperiment(pragmatic_listener, human_utterances)
    res = pd.DataFrame(results)
    res["alphaS"] = alpha
    alphaTestResults.append(res)

alphaTest = pd.concat(alphaTestResults)
```

```
AlphaS: 1.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 2.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 3.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 4.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 5.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 6.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 7.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 8.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
```

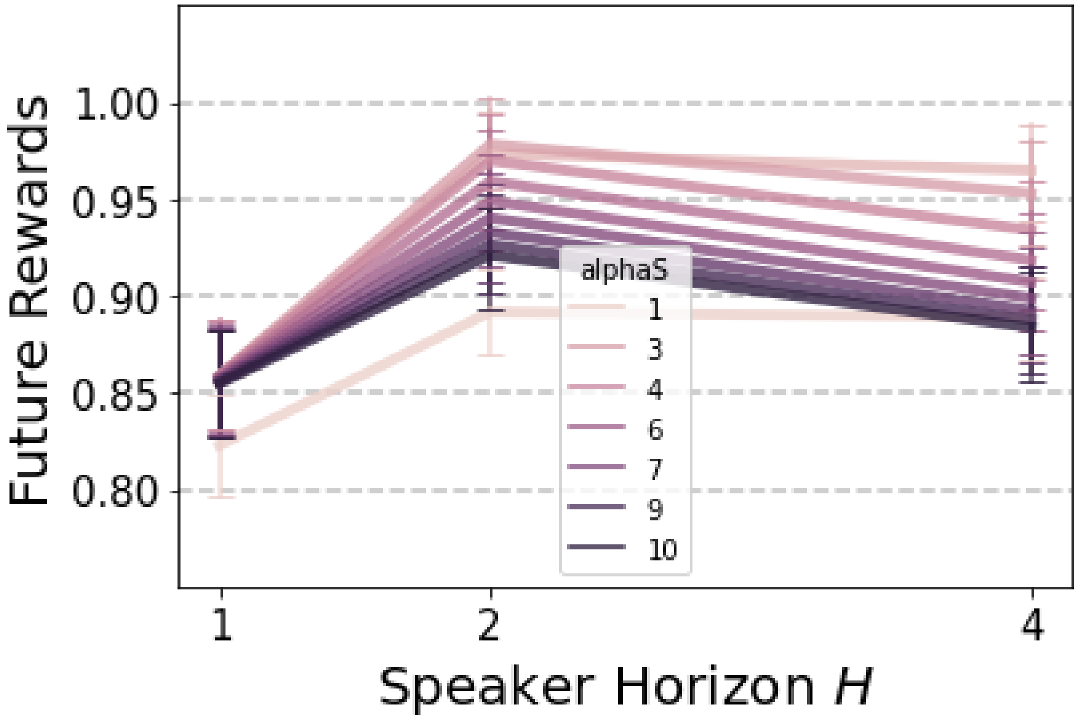
```
AlphaS: 9.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
AlphaS: 10.
Horizon 1: 939 utterances.
Horizon 2: 917 utterances.
Horizon 4: 916 utterances.
```

[63]:

```
alphaTestToPlot = alphaTest

sns.lineplot(data=alphaTestToPlot, x='horizon', y='pragmatic_aligned',␣
 ↪hue='alphaS', err_style='bars', err_kws={"capsize": 5},
             linewidth=4, alpha=.8)

ys = [.8, .85, .9, .95, 1]
for y in ys:
    plt.axhline(y, alpha=.25, linestyle='--', zorder=0, c='k')
plt.yticks(ys, fontsize=15)
plt.xticks([1, 2, 4], fontsize=15)

plt.ylim(.75, 1.05)

plt.xlabel("Speaker Horizon $H$", fontsize=20)
plt.ylabel("Future Rewards", fontsize=20)
```

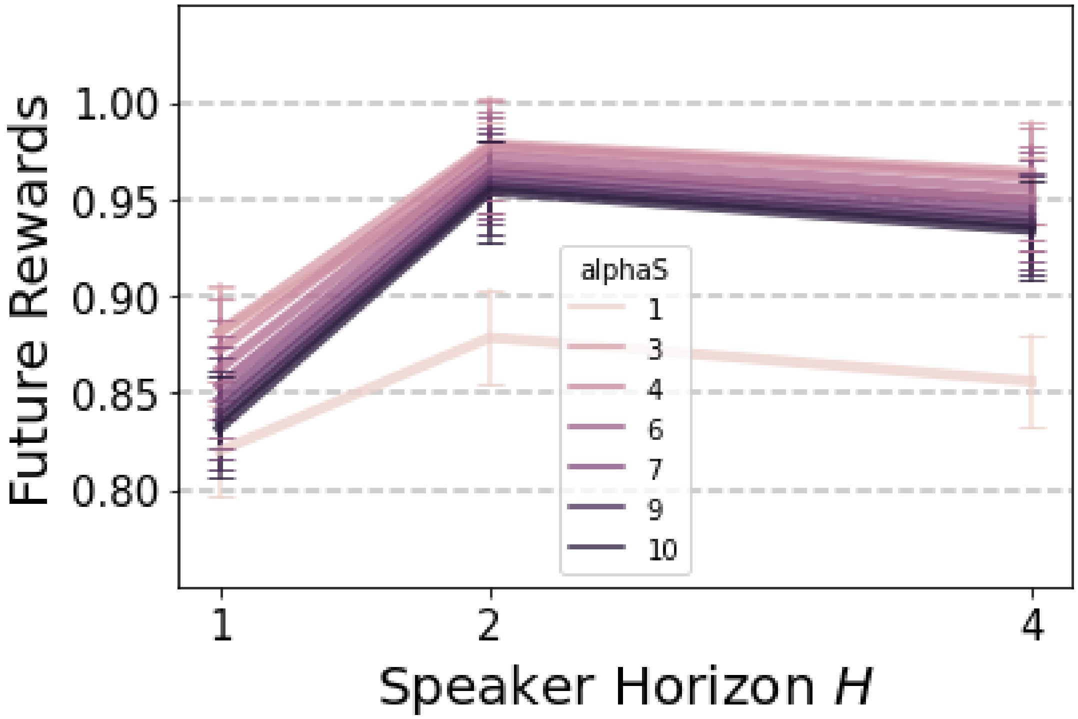
[63]:

```
Text(0, 0.5, 'Future Rewards')
```



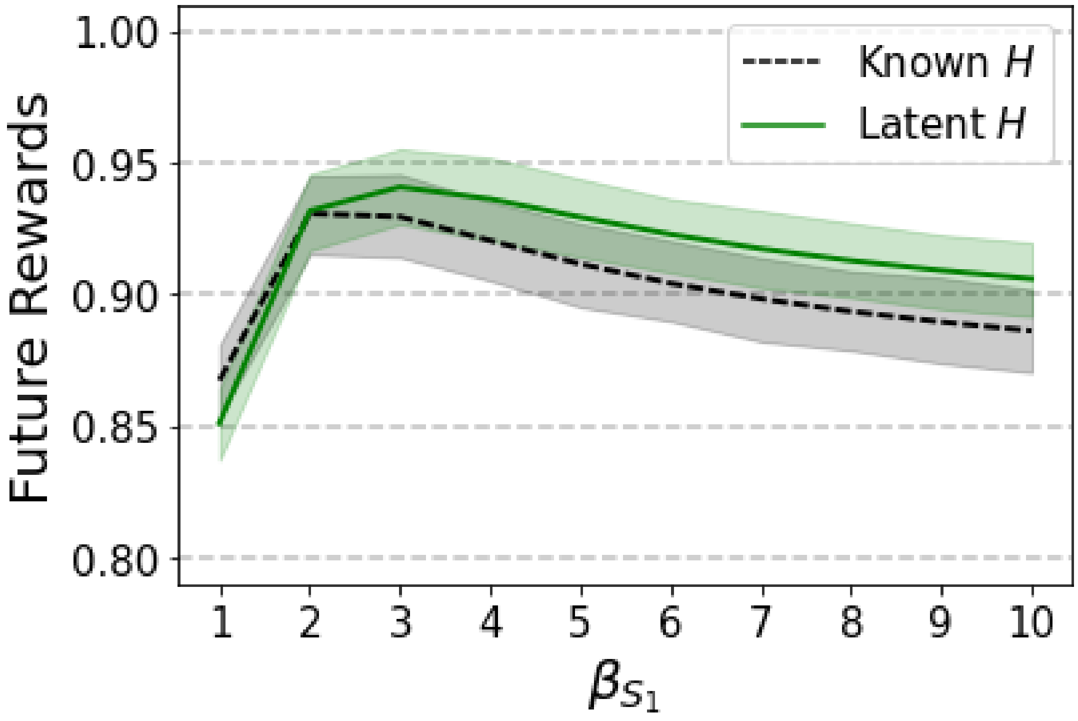
```
[64]: sns.lineplot(data=alphaTestToPlot, x='horizon', y='pragmatic_uncertain',␣
       ↪hue='alphaS', err_style='bars', err_kws={"capsize": 5},
                   linewidth=4, alpha=.8)

      ys = [.8, .85, .9, .95, 1]
      for y in ys:
          plt.axhline(y, alpha=.25, linestyle='--', zorder=0, c='k')
      plt.yticks(ys, fontsize=15)
      plt.xticks([1, 2, 4], fontsize=15)

      plt.ylim(.75, 1.05)

      plt.xlabel("Speaker Horizon $H$", fontsize=20)
      plt.ylabel("Future Rewards", fontsize=20)
```

```
[64]: Text(0, 0.5, 'Future Rewards')
```



[65]:
```
melted = alphaTestToPlot.melt(id_vars=["alphaS"],
                              value_vars=["pragmatic_aligned",␣
 ↪"pragmatic_uncertain"],
                              var_name="speaker", value_name="rewards")

rename_dict = {"pragmatic_aligned": "Known $H$", "pragmatic_uncertain": "Latent␣
 ↪$H$"}
melted["speaker"] = melted.speaker.apply(lambda x: rename_dict[x])
```

[66]:
```
sns.lineplot(data=melted, x='alphaS', y='rewards', hue='speaker',
             hue_order=["Known $H$", "Latent $H$"],
             palette=['k', 'g'],
              linewidth=2, style='speaker', dashes=[(3,1), ''])

plt.legend(loc='best', fontsize=15)

ys = [.8, .85, .9, .95, 1]
for y in ys:
    plt.axhline(y, alpha=.25, linestyle='--', zorder=0, c='k')
plt.yticks(ys, fontsize=15)
plt.xticks(range(1, 11), fontsize=15);

plt.ylabel("Future Rewards", fontsize=20)
plt.xlabel(r"$\beta_{S_1}$", fontsize=20);
```



[67]:
```
print(alphaTestToPlot.groupby(["alphaS"])[["pragmatic_aligned",␣
↪"pragmatic_uncertain"]].mean().round(4).style.to_latex())
```

```
\begin{tabular}{lrr}
 & pragmatic_aligned & pragmatic_uncertain \\
alphaS &  &  \\
1 & 0.867500 & 0.851200 \\
2 & 0.930800 & 0.931800 \\
3 & 0.929500 & 0.940800 \\
4 & 0.920500 & 0.936300 \\
5 & 0.911600 & 0.929400 \\
6 & 0.904300 & 0.922900 \\
7 & 0.898400 & 0.917400 \\
8 & 0.893600 & 0.912900 \\
9 & 0.889700 & 0.909200 \\
10 & 0.886300 & 0.906000 \\
\end{tabular}
```

[68]:
```
alpha_three_known_h = alphaTestToPlot[alphaTestToPlot.alphaS ==␣
↪3]["pragmatic_aligned"]
```

```
alpha_two_known_h = alphaTestToPlot[alphaTestToPlot.alphaS ==␣
 ↪2]["pragmatic_aligned"]

stats.ttest_rel(alpha_three_known_h, alpha_two_known_h)
```

```
[68]: Ttest_relResult(statistic=-1.696996750381101, pvalue=0.0898095767898488)
```