Analyzing the Logic Building Process of Large Language Neural Network Models

August 22, 2022

1 Introduction

This Jupyter notebook is a condensed summary of Meilong Zhang's summer 2022 SURF project on analyzing the logic-building processes in large language neural network models. The notebook demonstrates the entire research pipeline, covering dataset generation, network prompt engineering, bayesian analysis, and data analyses/visualization. To run the notebook, please make sure to pull or download from https://github.com/meilongzhang/concept-synthesis for the necessary packages and dependencies.

1.1 Project Abstract and Background Information:

Large language models fine-tuned on code have demonstrated proficiency in quantitative reasoning (Codex MIT paper, Google Minerva), competitive programming, among other tasks generally considered to be "intelligent." While evaluating the performance of large language models is important, we seek to expand discussion on the "how" of model learning. Specifically, we conduct a series of cognitive neuroscience-based "blicket" experiments on large language models (Codex, CodeT5) to analyze the patterns of logic-building employed within. We also relate these patterns to previous literature involving human cognition, manifested in human logical biases, such as a tendency towards simplicity and preference for conjunction-based rule-learning.

Traditional machine learning seeks to replicate results given sets of inputs and outputs. In a simple example of classifying apples and oranges, traditional machine learning would like to take in a new image and identify whether it is an image of an apple or orange. Program induction, on the other hand, seeks to clearly define patterns in input and output pairs. In the same example of apples and oranges, program induction would output the thought process behind saying whether an object is an apple or an orange.

The foremost approach to program induction has been through solver programs, which are traditional, deterministic software programs, written with domain knowledge. Programs written in this fashion are complex, and maladaptive to other unseen domains.

However, in recent years, advancements in natural language processing have enabled novel breakthroughs at the intersection of these two areas. By training language models on code, these models are now able to perform completions of code, translate between code and natural language, and write entire programs from natural language prompts. Some additional findings demonstrate the ability of these models to reason mathematically and logically.

While the majority of these analyses focus on the accuracy of the model (percentage of math problems, programming questions, logic puzzles answered correctly), my project this summer has focused on how these models build the internal logic necessary for reasoning on logical and mathematical problems. Specifically, I look at the logic-building process through the human cognitive-neuroscience lens, and investigate whether these language models share common logical biases with humans. To this end, I use a blicket task environment, do experimentation using the OpenAI Codex model, and utilize LOTLib3 - a tool designed in my lab - for Bayesian analysis, which is shown to be a good approximate for human learning.

2 Setup

[1]:
```
%cd libraries
import LOTlib3
import os
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from random import sample, randrange, choices
import json
from LOTlib3.Hypotheses.LOTHypothesis import LOTHypothesis
from LOTlib3.DataAndObjects import FunctionData, Obj
from LOTlib3.DefaultGrammars import DNF
from LOTlib3.Miscellaneous import q, random, qq
from LOTlib3.Grammar import Grammar
from LOTlib3.Hypotheses import FunctionHypothesis, Hypothesis
from LOTlib3.Samplers.MetropolisHastings import MetropolisHastingsSampler
from LOTlib3 import break_ctrlc
from LOTlib3.TopN import TopN
from LOTlib3.Hypotheses.Priors.RationalRules import RationaRulesPrior
from LOTlib3.Hypotheses.Likelihoods.BinaryLikelihood import BinaryLikelihood
import itertools
import openai
import seaborn as sns
import time
import plotly.graph_objects as go
import plotly.express as px
import re
from scipy.special import logsumexp
```

```
/Users/meilongzhang/knightlab/codet5/libraries
```

```
/opt/anaconda3/lib/python3.7/site-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm
```

3 LOTlib3 Setup and Dataset Generation

In the blicket environment, participants are given examples of a (color, shape) pair, as well as a boolean value indicating whether that pair belongs in the target rule domain. The target rule remains hidden from the participant, but is implicitly learned through trial and error. In my experiment, I am using a set of 3 colors and 3 shapes, and constricting rules to only those constructed from boolean operations. Previous literature on blicket tasks and similar concept learning tasks pinpoint several fundamental human logical biases, such as a preference for simplicity and brevity, preference for rules involving conjunctions over rules involving disjunctions, as well as the win-stay-lose-shift heuristic, where humans tend to stick with a internal hypothesis until they make a conflicting observation. My results, to be shown later, demonstrate that Codex shares these same biases.

[2]:
```
DEFAULT_FEATURE_WEIGHT = 5
grammar = Grammar()
grammar.add_rule('START', '', ['DISJ'], 1.0)
grammar.add_rule('START', '', ['PRE-PREDICATE'], DEFAULT_FEATURE_WEIGHT)
grammar.add_rule('START', 'True', None, DEFAULT_FEATURE_WEIGHT)
grammar.add_rule('START', 'False', None, DEFAULT_FEATURE_WEIGHT)

# Disjunctions
grammar.add_rule('DISJ', '',      ['CONJ'], 1.0)
grammar.add_rule('DISJ', '',      ['PRE-PREDICATE'], DEFAULT_FEATURE_WEIGHT)
grammar.add_rule('DISJ', '(%s or %s)',  ['PRE-PREDICATE', 'DISJ'], 1.0)

# Conjunctions
grammar.add_rule('CONJ', '',      ['PRE-PREDICATE'], DEFAULT_FEATURE_WEIGHT)
grammar.add_rule('CONJ', '(%s and %s)', ['PRE-PREDICATE', 'CONJ'], 1.0)

# Pre-Predicates
grammar.add_rule('PRE-PREDICATE', '(not (%s))', ['PREDICATE'],␣
↪DEFAULT_FEATURE_WEIGHT)
grammar.add_rule('PRE-PREDICATE', '',     ['PREDICATE'], DEFAULT_FEATURE_WEIGHT)

# Predicates
grammar.add_rule('PREDICATE', "x['color'] == %s", ['COLOR'], 1.0)
grammar.add_rule('PREDICATE', "x['shape'] == %s", ['SHAPE'], 1.0)

# Colors
grammar.add_rule('COLOR', q('red'), None, 1.0)
grammar.add_rule('COLOR', q('blue'), None, 1.0)
grammar.add_rule('COLOR', q('green'), None, 1.0)

# Shapes
grammar.add_rule('SHAPE', q('square'), None, 1.0)
grammar.add_rule('SHAPE', q('circle'), None, 1.0)
grammar.add_rule('SHAPE', q('triangle'), None, 1.0);
```

[3]:
```
class MyHypothesis(RationaRulesPrior, BinaryLikelihood, LOTHypothesis):
    def __init__(self, **kwargs):
        LOTHypothesis.__init__(self, grammar=grammar, **kwargs)
        self.rrAlpha=2.0
```

Below is the complete list of possible (color, shape) pairs.

[4]:
```
colors = ['red', 'blue', 'green']
shapes = ['circle', 'square', 'triangle']
all_stimuli = []

for color in colors:
    for shape in shapes:
        all_stimuli.append({'shape':shape, 'color':color})

all_stimuli
```

[4]:
```
[{'shape': 'circle', 'color': 'red'},
 {'shape': 'square', 'color': 'red'},
 {'shape': 'triangle', 'color': 'red'},
 {'shape': 'circle', 'color': 'blue'},
 {'shape': 'square', 'color': 'blue'},
 {'shape': 'triangle', 'color': 'blue'},
 {'shape': 'circle', 'color': 'green'},
 {'shape': 'square', 'color': 'green'},
 {'shape': 'triangle', 'color': 'green'}]
```

Generate all possible configurations of True and False values, for a total of $2^9 = 512$ possible results arrays.

[5]:
```
l = [True, False]
all_results = [list(i) for i in itertools.product(l, repeat=9)]
```

For each result array, we use the Metropolis Hastings Sampler to find a rule that correctly describes the 9 perceived inputs as True or False. Firstly, we search 10000 steps, and keep track of the Top 10 hypotheses. The top 10 hypotheses are then evaluated based on accuracy on the 9 perceived inputs, and the hypotheses with the highest accuracy ($accuracy == max(accuracies)$) are kept. Of the hypotheses with the highest accuracy on the "training set", the one with highest likelihood and posterior score is recorded in the dataset as the true underlying function.

The code implementation is in the cell below. Usual runtime is about 4 to 5 hours, so please skip running this cell.

[ ]:
```
with open('../data/revised_codex_prompts.json', 'w') as out:
    da = []
    for results in all_results:
    #for results in all_results:
        print(results)
```

```
        objs = [FunctionData(input=[all_stimuli[i]], output=results[i], alpha=0.
↪999) for i in range(9)]
        print(objs)
        hypo = MyHypothesis()
        top = TopN(N=10)
        print(f"sampling {all_results.index(results)}")
        for h in MetropolisHastingsSampler(hypo, objs, steps=10000):
            top << h

        codes = []
        posts = []
        priors = []
        likelihoods = []
        for h in top:
            codes.append(qq(h))
            posts.append(h.posterior_score)
            priors.append(h.prior)
            likelihoods.append(h.likelihood)

        corrects = []
        for code in codes:
            exec(f"def classify(x): return {code[11:len(code)-1]}")
            correct = 0
            for i in range(len(all_stimuli)):
                correct += classify(all_stimuli[i]) == results[i]
            corrects.append(correct)
        print(corrects)

        best_indices = [i for i in range(len(corrects)) if corrects[i] ==␣
↪max(corrects)]
        data = {}
        print(codes[posts.index(max([posts[i] for i in best_indices]))])
        best_index = posts.index(max([posts[i] for i in best_indices]))
        data["code"] = str(codes[best_index])
        data["accuracy"] = str(corrects[best_index] / 9)
        data["stims"] = str(all_stimuli)
        data["results"] = str(results)
        da.append(data)

    out.write(json.dumps(da))
out.close()
```

Below is the resulting dataset. The "code" column contains the best possible hypothesis found by LOTlib3, evaluated by accuracy on inputs and posterior score. The "accuracy" column contains the accuracy of the hypothesis on the 9 inputs. The "stims" column contains the 9 (color, shape) inputs. "Results" contains the True or False value corresponding to each (color, shape) pair.

[6]:
```python
test_prompts = pd.read_json("../data/revised_codex_prompts_2.json")
test_prompts
```

[6]:
```
                                                  code  accuracy  \
0                                   "lambda x: True"       1.0
1    "lambda x: ((not (x['color'] == 'green')) or (…      1.0
2    "lambda x: ((not (x['color'] == 'green')) or (…      1.0
3    "lambda x: ((not (x['color'] == 'green')) or x…      1.0
4    "lambda x: ((not (x['color'] == 'green')) or (…      1.0
..                                                 …        …
507  "lambda x: (x['color'] == 'green' and x['shape…      1.0
508  "lambda x: ((not (x['shape'] == 'circle')) and…      1.0
509  "lambda x: (x['color'] == 'green' and x['shape…      1.0
510  "lambda x: (x['color'] == 'green' and x['shape…      1.0
511                                "lambda x: False"       1.0

                                                 stims  \
0    [{'shape': 'circle', 'color': 'red'}, {'shape'…
1    [{'shape': 'circle', 'color': 'red'}, {'shape'…
2    [{'shape': 'circle', 'color': 'red'}, {'shape'…
3    [{'shape': 'circle', 'color': 'red'}, {'shape'…
4    [{'shape': 'circle', 'color': 'red'}, {'shape'…
..                                                 …
507  [{'shape': 'circle', 'color': 'red'}, {'shape'…
508  [{'shape': 'circle', 'color': 'red'}, {'shape'…
509  [{'shape': 'circle', 'color': 'red'}, {'shape'…
510  [{'shape': 'circle', 'color': 'red'}, {'shape'…
511  [{'shape': 'circle', 'color': 'red'}, {'shape'…

                                               results
0    [True, True, True, True, True, True, True, Tru…
1    [True, True, True, True, True, True, True, Tru…
2    [True, True, True, True, True, True, True, Fal…
3    [True, True, True, True, True, True, True, Fal…
4    [True, True, True, True, True, True, False, Tr…
..                                                 …
507  [False, False, False, False, False, False, Tru…
508  [False, False, False, False, False, False, Fal…
509  [False, False, False, False, False, False, Fal…
510  [False, False, False, False, False, False, Fal…
511  [False, False, False, False, False, False, Fal…

[512 rows x 4 columns]
```

4 Codex Induction

To modify the task into a proper format for the language model as well as to maximize the accuracy of the model, I use a variety of prompt engineering techniques. Prompt engineering is an effective and resource-friendly way to improve language model performance, and is competitive with the traditional fine-tuning approach. Techniques include chain-of-thought prompting, few-shot prompting, and majority sampling. These are demonstrated in the example prompt later on in the section. The language model is fed a string representation of multiple experiment trials, which provides the necessary context to generalize to new observations.

Below we run the program induction through Codex.

[7]:
```
openai.api_key = "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"
```

Below is an example of a Codex API call. "code-davinci-002" indicates specifies the Codex engine we are using. Max tokens indicate the maximum length of the completion sequence, temperature indicates the amount of variability in selecting between the "best" completion and other completions. In our approach, we ask Codex to generate 50 completions for any given prompt, and return the 5 best completions according to Codex's evaluation metric (log probability of tokens).

We then construct 9 assert statement unit tests from the inputted stimuli and corresponding results, and evaluate the 5 completions on these unit tests. The completion that performs best is kept.

The evaluation process is broken into two stages as a higher log probability of tokens does not necessarily indicate better performance on unit tests.

To ensure creativity in Codex generation and diversity in completion sequences, we use a temperature of 0.4.

Example Codex API Call:

[ ]:
```
completion = openai.Completion.create(
  engine="code-davinci-002",
  prompt=new_prompt,
  max_tokens=150,
  temperature= 0.4,
  best_of = 50,
  n= 5
)
```

run_tests takes in a Codex API call result and the corresponding unit tests. For the call result, iterate through the completions to find the best-performing completion on the unit tests. We then pass back the code of the best completion, corresponding accuracy, and passed and failed tests.

[8]:
```
def run_tests(i, completion, tests):
    '''
    Run assert unit tests on a Codex-generated function.

    Parameters:
        i (int): Test label for checking progress
        completion: Codex-generated completion
```

```
        tests (list): List of assert statement strings

    Returns:
        acc (float): Accuracy corresponding to the best completion
        code (str): The completion that is most accurate on the unit tests
        passed (list): Unit tests the best completion passes
        failed (list): Unit tests the best completion fails
    '''
    accs = []
    failed = []
    passed = []
    codes = []
    for comp in completion['choices']:
        failed_tests = []
        passed_tests = []
        gen_code = "def categorize(color, shape):\n"
        try:
            gen_code += f"\t{comp['text'].strip().splitlines()[0]}"
        except IndexError as e:
            continue
        print(f"Test #{i}: {gen_code}:")
        codes.append(gen_code)
        try:
            exec(gen_code)
        except SyntaxError as e:
            continue
        num = len(tests)
        num_correct = 0
        for test in tests:
            try:
                exec(test)
                num_correct += 1
                passed_tests.append(test)
            except:
                failed_tests.append(test)
            print(f"Accuracy: {num_correct/num}\n\n")
            accs.append(num_correct/num)
            failed.append(failed_tests)
            passed.append(passed_tests)
        print(f"Highest Accuracy: {max(accs)}")
    return (max(accs), codes[accs.index(max(accs))], passed[accs.
↪index(max(accs))], failed[accs.index(max(accs))])
```

[9]:
```
def apply_rule_t(rule, stimuli):
    '''
    Apply rule to each stimulus to generate correct results.
```

```
    Parameters:
        rule (str): String representation of lambda function
        stimuli (list): List of dictionaries (stimuli)

    Returns:
        results (list): List of boolean values
    '''
    assert type(rule) == str
    rule = eval(rule)
    results = []
    for stim in stimuli:
        results.append(rule(stim))
    return results
```

get_asserts takes in the list of (color, shape) inputs and the corresponding results, and generates the new prompt to feed into the Codex API as well as the assert unit tests through which we evaluate the API completion.

```
[10]: def get_asserts(stims, results):
    '''
    Generate the new prompt and unit tests.

    Parameters:
        stims (list): List of stimuli
        results (list): List of corresponding results

    Returns:
        new_prompt (str): Prompt to be passed in to OpenAI API call
        tests (list): List of assert unit tests
    '''
    new_prompt = f"def categorize(color: str, shape: str) -> bool:\n"
    new_prompt += f"\t\"\"\"Determine what colors and shapes are part of the␣
↪category.\n"
    tests = []
    for j in range(len(stims)):
        assert_statement = f"assert categorize('{stims[j]['color']}',␣
↪'{stims[j]['shape']}') == {results[j]}"
        new_prompt += f"\t>>> categorize('{stims[j]['color']}',␣
↪'{stims[j]['shape']}')"
        new_prompt += "\n"
        new_prompt += f"\t{results[j]}"
        new_prompt += "\n"
        tests.append(assert_statement)
    new_prompt += "\t\"\"\""
    new_prompt = prompt + new_prompt
    return (new_prompt, tests)
```

In prior experimentation, we discovered that a few-shot approach prompts out better completions from Codex compared to a one-shot approach. To cover the hypothesis space as much as possible, we are using manually designed few-shot examples as shown below. This "pre-prompt" is appended in front of every new generated prompt to create the few-shot design.

We deliberately choose four examples of varying concept complexity to encourage diversity in Codex generations.

[11]:
```
one = "lambda x: (x['color'] == 'green')"
two = "lambda x: (x['color'] == 'red' and x['shape'] == 'square')"
more = "lambda x: (x['color'] == 'blue' or (not (x['shape'] == 'triangle')))"
longer = "lambda x: ((x['color'] == 'green' and x['shape'] == 'circle') or␣
 ↪((not (x['color'] == 'green')) and x['shape'] == 'square'))"
prompt_examples = [one, two, more, longer]

onecs = "(color == 'green')"
twocs = "(color == 'red' and shape == 'square')"
morecs = "(color == 'blue' or (not (shape == 'triangle')))"
longercs = "((color == 'green' and shape == 'circle') or ((not (color ==␣
 ↪'green')) and shape == 'square'))"
code_snippets = [onecs, twocs, morecs, longercs]
```

The pre-prompt is printed by the next cell.

[12]:
```
prompt = f""
i = 0
for p in prompt_examples:
    prompt += "def categorize(color: str, shape: str) -> bool:\n"
    prompt += "\t\"\"\" Determine what colors and shapes are part of the␣
 ↪category.\n"
    for s in all_stimuli:
        result = apply_rule_t(p, [s])[0]
        statement = f"\t>>> categorize('{s['color']}', '{s['shape']}')"
        statement += "\n"
        statement += f"\t{result}"
        statement += "\n"
        prompt += statement
    answer = f"\t\"\"\"\n\treturn {code_snippets[i]}\n\n"
    prompt += answer
    i += 1
print(prompt)
```

```
def categorize(color: str, shape: str) -> bool:
        """ Determine what colors and shapes are part of the category.
        >>> categorize('red', 'circle')
        False
        >>> categorize('red', 'square')
        False
```

```
        >>> categorize('red', 'triangle')
        False
        >>> categorize('blue', 'circle')
        False
        >>> categorize('blue', 'square')
        False
        >>> categorize('blue', 'triangle')
        False
        >>> categorize('green', 'circle')
        True
        >>> categorize('green', 'square')
        True
        >>> categorize('green', 'triangle')
        True
        """
        return (color == 'green')

def categorize(color: str, shape: str) -> bool:
        """ Determine what colors and shapes are part of the category.
        >>> categorize('red', 'circle')
        False
        >>> categorize('red', 'square')
        True
        >>> categorize('red', 'triangle')
        False
        >>> categorize('blue', 'circle')
        False
        >>> categorize('blue', 'square')
        False
        >>> categorize('blue', 'triangle')
        False
        >>> categorize('green', 'circle')
        False
        >>> categorize('green', 'square')
        False
        >>> categorize('green', 'triangle')
        False
        """
        return (color == 'red' and shape == 'square')

def categorize(color: str, shape: str) -> bool:
        """ Determine what colors and shapes are part of the category.
        >>> categorize('red', 'circle')
        True
        >>> categorize('red', 'square')
        True
        >>> categorize('red', 'triangle')
        False
```

```
        >>> categorize('blue', 'circle')
        True
        >>> categorize('blue', 'square')
        True
        >>> categorize('blue', 'triangle')
        True
        >>> categorize('green', 'circle')
        True
        >>> categorize('green', 'square')
        True
        >>> categorize('green', 'triangle')
        False
        """
        return (color == 'blue' or (not (shape == 'triangle')))

def categorize(color: str, shape: str) -> bool:
        """ Determine what colors and shapes are part of the category.
        >>> categorize('red', 'circle')
        False
        >>> categorize('red', 'square')
        True
        >>> categorize('red', 'triangle')
        False
        >>> categorize('blue', 'circle')
        False
        >>> categorize('blue', 'square')
        True
        >>> categorize('blue', 'triangle')
        False
        >>> categorize('green', 'circle')
        True
        >>> categorize('green', 'square')
        False
        >>> categorize('green', 'triangle')
        False
        """
        return ((color == 'green' and shape == 'circle') or ((not (color ==
'green')) and shape == 'square'))
```

Along with running all 9 stimuli through Codex, we also investigate the ability of Codex to generalize through fewer examples. To this point, we have Codex generalize on the range of 1 to 9 seen examples, in an effort to mimic the learning curves observed in human subjects completing the same task. Seen examples are examples to which Codex knows the correct classification (True or False).

For each rule and each amount of seen stimuli (1 to 9), we store the best completion generated by Codex, its corresponding accuracy, passed unit tests, failed unit tests, number of stimuli seen, stimuli seen, rule number, as well as generated code and true code.

Runtime of this cell takes several hours due to Codex-imposed rate limits. As a workaround to the 20 completions/min and 150000 tokens/min rate limits, we force a sleep after every completion.

[ ]:
```
dataset = pd.DataFrame()
for i in range(len(test_prompts)):
    actual_code = test_prompts['code'][i]
    actual_acc = test_prompts['accuracy'][i]
    stims = eval(test_prompts['stims'][i])
    results = eval(test_prompts['results'][i])

    for j in range(1, len(stims) + 1):
        df = pd.DataFrame()
        stimset = stims[:j]
        resultset = results[:j]
        new_prompt, _ = get_asserts(stimset, resultset)
        _, tests = get_asserts(stims, results)
        completion = openai.Completion.create(
            engine="code-davinci-002",
            prompt=new_prompt,
            max_tokens=150,
            temperature= 0.4,
            best_of= 50,
            n = 5
        )
        time.sleep(10)

        acc, gen_code, passed, failed = run_tests(i+1, completion, tests)
        gen_code_concat = gen_code[38:len(gen_code)]
        tr_code_concat = actual_code[11:len(actual_code) - 1].
 ↪replace("x['shape']", "shape").replace("x['color']", "color").replace("==",␣
 ↪" == ")
        df['Problem_num'] = [i+1]
        df['gen_accuracy'] = [acc]
        df['tr_accuracy'] = [actual_acc]
        df['tr_code_concat'] = [tr_code_concat]
        df['gen_code_concat'] = [gen_code_concat]
        df['tr_code_size'] = [len(tr_code_concat)]
        df['gen_code_size'] = [len(gen_code_concat)]
        df['num_stims_seen'] = [j]
        df['stims_seen'] = [stimset]
        df['passed_tests'] = [passed]
        df['failed_tests'] = [failed]
        df['tr_code_full'] = [actual_code]
        df['gen_code_full'] = [gen_code]
        dataset = pd.concat([dataset, df])
```

The resulting dataset:

[13]:
```python
data = pd.read_csv("../data/full_output.csv").drop("Unnamed: 0", axis=1)
data
```

[13]:
```
      Problem_num  accuracy                              tr_code_concat  \
0               1  0.555556                                        True
1               1  0.555556                                        True
2               1  1.000000                                        True
3               1  1.000000                                        True
4               1  1.000000                                        True
…             …       …                                           …
1192          133  0.777778  (shape == 'triangle' and color == 'green')
1193          133  0.888889  (shape == 'triangle' and color == 'green')
1194          133  0.888889  (shape == 'triangle' and color == 'green')
1195          133  0.888889  (shape == 'triangle' and color == 'green')
1196          133  0.777778  (shape == 'triangle' and color == 'green')

                             gen_code_concat  true_code_size  gen_code_size  \
0         (color == 'red' or shape == 'circle')               4             37
1         (color == 'red' or shape == 'square')               4             37
2                                       True               4              4
3                                       True               4              4
4                                       True               4              4
…                                          …             …             …
1192  (color == 'blue' and shape == 'square')              42             39
1193                                   False              42              5
1194                                   False              42              5
1195                                   False              42              5
1196                     (shape == 'triangle')              42             21

      num_stims_seen                                    stims_seen  \
0                  1  [{'shape': 'circle', 'color': 'red', 'alpha': …
1                  2  [{'shape': 'circle', 'color': 'red', 'alpha': …
2                  3  [{'shape': 'circle', 'color': 'red', 'alpha': …
3                  4  [{'shape': 'circle', 'color': 'red', 'alpha': …
4                  5  [{'shape': 'circle', 'color': 'red', 'alpha': …
…                …                                           …
1192               5  [{'shape': 'circle', 'color': 'red', 'alpha': …
1193               6  [{'shape': 'circle', 'color': 'red', 'alpha': …
1194               7  [{'shape': 'circle', 'color': 'red', 'alpha': …
1195               8  [{'shape': 'circle', 'color': 'red', 'alpha': …
1196               9  [{'shape': 'circle', 'color': 'red', 'alpha': …

                                          passed_tests  \
0     ["assert categorize('red', 'circle') == True",…
1     ["assert categorize('red', 'circle') == True",…
```

```
2     ["assert categorize('red', 'circle') == True",…
3     ["assert categorize('red', 'circle') == True",…
4     ["assert categorize('red', 'circle') == True",…
…                                                  …
1192  ["assert categorize('red', 'circle') == False"…
1193  ["assert categorize('red', 'circle') == False"…
1194  ["assert categorize('red', 'circle') == False"…
1195  ["assert categorize('red', 'circle') == False"…
1196  ["assert categorize('red', 'circle') == False"…

                                          failed_tests  \
0     ["assert categorize('blue', 'square') == True"…
1     ["assert categorize('blue', 'circle') == True"…
2                                                   []
3                                                   []
4                                                   []
…                                                  …
1192  ["assert categorize('blue', 'square') == False…
1193  ["assert categorize('green', 'triangle') == Tr…
1194  ["assert categorize('green', 'triangle') == Tr…
1195  ["assert categorize('green', 'triangle') == Tr…
1196  ["assert categorize('red', 'triangle') == Fals…

                                          tr_code_full  \
0                                  "lambda x: True"
1                                  "lambda x: True"
2                                  "lambda x: True"
3                                  "lambda x: True"
4                                  "lambda x: True"
…                                                  …
1192  "lambda x: (x['shape']=='triangle' and x['colo…
1193  "lambda x: (x['shape']=='triangle' and x['colo…
1194  "lambda x: (x['shape']=='triangle' and x['colo…
1195  "lambda x: (x['shape']=='triangle' and x['colo…
1196  "lambda x: (x['shape']=='triangle' and x['colo…

                                         gen_code_full  tr_domain  \
0     def categorize(color, shape):\n\treturn (color…         9
1     def categorize(color, shape):\n\treturn (color…         9
2          def categorize(color, shape):\n\treturn True          9
3          def categorize(color, shape):\n\treturn True          9
4          def categorize(color, shape):\n\treturn True          9
…                                                  …        …
1192  def categorize(color, shape):\n\treturn (color…         1
1193      def categorize(color, shape):\n\treturn False          1
1194      def categorize(color, shape):\n\treturn False          1
1195      def categorize(color, shape):\n\treturn False          1
```

```
1196  def categorize(color, shape):\n\treturn (shape…          1

      gen_domain                                      gen_reshaped
0              5  "lambda x: (x['color'] == 'red' or x['shape'] …
1              5  "lambda x: (x['color'] == 'red' or x['shape'] …
2              9                                "lambda x: True"
3              9                                "lambda x: True"
4              9                                "lambda x: True"
…            …                                                 …
1192           1  "lambda x: (x['color'] == 'blue' and x['shape'…
1193           0                               "lambda x: False"
1194           0                               "lambda x: False"
1195           0                               "lambda x: False"
1196           3            "lambda x: (x['shape'] == 'triangle')"

[1197 rows x 15 columns]
```

5 Evaluating Codex Outputs Using Bayesian Network Analysis

The next step is to convert Codex generated code (string outputs) into the grammar structure compatible with LOTlib3 (FunctionNode objects). I recursively build the FunctionNode tree relationship from a generated string, creating a new MyHypothesis object that adheres to the specified grammar rules. Most of the methods below are helper methods, with the most important method being convertToNode, which ties everything together.

[14]:
```
class Clause:
    name = ''
    leftChild = None
    rightChild = None
    parent = None
    node = None

    def __init__(self, n):
        self.name = n
        if (n != 'sentinel'):
            self.parent = self.getSentinel()

    def getSentinel(self):
        return Clause('sentinel')

    def getName(self):
        return self.name

    def setNode(self, no):
        self.node = no
```

```
    def getNode(self):
        return self.node

    def getChildren(self):
        lst = []
        if (self.leftChild != None):
            lst.append(self.leftChild)
        if (self.rightChild != None):
            lst.append(self.rightChild)
        return lst

    def isLeaf(self):
        return self.leftChild == None and self.rightChild == None
```

[15]:
```
def makeClauseList(code, startIndex):
    """
    Separate code string into 'clauses': logical operators (and, or, not)
    as well as substrings enclosed by parentheses. Only separates the first
    layer, which means combinations of clauses enclosed by parentheses
    ie (x['color'] == 'blue' or x['color'] == 'green') is treated as one clause.
    Calls convertToClause before returning to convert strings in Clause objects.

    Parameters:
        code (str): String of code.
        startIndex (int): Index at which to begin parsing the code string.

    Returns:
        clauseList (list): List of clauses that represent the code string.
    """
    clauseList = []
    i = startIndex
    substr = ''
    while i < len(code):
        if (code[i] == '('):
            clause, i = makeClause(code, i+1, 1)
            clauseList.append(clause)

        elif (code[i] == 'n' and code[i+1] == 'o'):
            clauseList.append("not")
            i = i+3

        elif (code[i].isspace() and code[i+1] == 'a'):
            if (substr != ''):
                clauseList.append(substr)
                substr = ''
            clauseList.append("and")
            i = i+4
```

```
        elif (code[i].isspace() and code[i+1] == 'o'):
            if (substr != ''):
                clauseList.append(substr)
                substr = ''
            clauseList.append("or")
            i = i+3

        else:
            substr += code[i]

        i = i+1

    if (substr != ''):
        clauseList.append(substr)

    clauseList = convertToClause(clauseList)
    return clauseList


def makeClause(code, ind, num):
    """
    Parse code string until the corresponding close parentheses is found.

    Parameters:
        code (str): Code string.
        ind (int): Index at which to start parsing string.
        num (int): Number of open parentheses to account for. Default is 1,
                because finding an open parentheses leads to a makeClause␣
 ↪function call.

    Returns:
        clause (str): The substring enclosed by the initial open parentheses␣
 ↪and found close parentheses.
        ind (int): Index at which makeClauseList should resume parsing the␣
 ↪remainder of the string.
    """
    clause = ''
    while (num != 0):
        if (code[ind] == '('):
            num += 1
        elif (code[ind] == ')'):
            num -= 1
            if (num == 0):
                break
        clause += code[ind]
        ind += 1
```

```python
    return clause, ind


def convertToClause(lst):
    """
    Converts each clause string in a list into a clause object. Calls␣
 ↪createParent to establish
    parent/child relationships between clause objects.

    Parameters:
        lst (list): List containing clause strings.

    Returns:
        new_lst (list): List of clause objects.
    """
    new_lst = []
    for item in lst:
        clause = Clause(item)
        new_lst.append(clause)

    if (len(new_lst) > 1):
        return createParent(new_lst)
    return new_lst

def createParent(lst):
    """
    Assigns parent and left/right child relationships between clause objects in␣
 ↪a list.

    Parameters:
        lst (list): List of clause objects.

    Returns:
        lst (list): List of clause objects with parent/child relationships.
    """
    for i in range(len(lst)):
        if (lst[i].name == 'and' or lst[i].name == 'or'):
            if (lst[i-1].parent.getName() != 'sentinel'):
                lst[i].leftChild = lst[i-2]
                lst[i-2].parent = lst[i]
            else:
                lst[i].leftChild = lst[i-1]
                lst[i-1].parent = lst[i]
            lst[i].rightChild = lst[i+1]
            lst[i+1].parent = lst[i]
        elif (lst[i].name == 'not'):
            lst[i].leftChild = None
```

```
            lst[i].rightChild = lst[i+1]
            lst[i+1].parent = lst[i]
    return lst


def recurseClauseList(lst):
    """
    Recursively go through a list of clause objects, breaking each clause␣
↪object into indivisible clauses,
    while maintaining parent/child relationships between clauses. Creates␣
↪FunctionNode representations
    of indivisible clauses.

    For example, a clause object of (x['color'] == 'blue' or x['color'] ==␣
↪'green') will now be broken
    into [x['color'] == 'blue', or, x['color'] == 'green'].

    Parameters:
        lst (list): List of clause objects.

    Returns:
        None
    """
    for i in range(len(lst)):
        tst = makeClauseList(lst[i].getName(), 0)
        if len(tst) == 1: # is base clause
            if (lst[i].getName() == 'or'):
                node = LOTlib3.FunctionNode.FunctionNode(returntype='DISJ',␣
↪name='(%s or %s)', parent=None, args=None)
                lst[i].setNode(node)
            elif (lst[i].getName() == 'and'):
                node = LOTlib3.FunctionNode.FunctionNode(returntype='CONJ',␣
↪name='(%s and %s)', parent=None, args=None)
                lst[i].setNode(node)
            elif (lst[i].getName() == 'not'):
                node = LOTlib3.FunctionNode.
↪FunctionNode(returntype='PRE-PREDICATE', name='(not %s)', parent=None,␣
↪args=None)
                lst[i].setNode(node)
            elif (lst[i].getName() == 'True'):
                node = LOTlib3.FunctionNode.FunctionNode(returntype='START',␣
↪name='True', parent=None, args=None)
                lst[i].setNode(node)
            elif (lst[i].getName() == 'False'):
                node = LOTlib3.FunctionNode.FunctionNode(returntype='START',␣
↪name='False', parent=None, args=None)
```

```python
                lst[i].setNode(node)
            else:
                node = convert_predicate(lst[i].getName())
                lst[i].setNode(node)
        else:
            if (lst[i].parent.leftChild == lst[i]):
                rootNode = getRoot(tst)
                lst[i].parent.leftChild = rootNode
                rootNode.parent = lst[i].parent
            elif (lst[i].parent.rightChild == lst[i]):
                rootNode = getRoot(tst)
                lst[i].parent.rightChild = rootNode
                rootNode.parent = lst[i].parent
            recurseClauseList(tst) # need to continue recursing


def convert_predicate(code, par=None):
    """
    Convert an indivisible predicate clause into FunctionNode representation.

    Parameters:
        code (str): Code representation of indivisible predicate clause.
        par (FunctionNode): FunctionNode object that should be the parent of␣
↪the new created predicate FunctionNode.

    Returns:
        node (FunctionNode): FunctionNode object.
    """
    code = code.split(' ')
    if code[0] == "x['shape']":
        node = LOTlib3.FunctionNode.FunctionNode(returntype='PREDICATE',␣
↪name="x['shape'] == %s", parent=par, args=None)
        node2 = LOTlib3.FunctionNode.FunctionNode(returntype='SHAPE',␣
↪name=code[2], parent=node, args=None)
        node.args = [node2]
    elif code[0] == "x['color']":
        node = LOTlib3.FunctionNode.FunctionNode(returntype='PREDICATE',␣
↪name="x['color'] == %s", parent=par, args=None)
        node2 = LOTlib3.FunctionNode.FunctionNode(returntype='COLOR',␣
↪name=code[2], parent=node, args=None)
        node.args = [node2]
    elif code[0] == "not":
        node = convert_negation(code, None)
    return node

def connectTree(lst):
    """
```

```
    Create parent/child relationships between the FunctionNode objects, using␣
↪the corresponding clause objects
    as a guide.

    Parameters:
        lst (list): List of clauses.

    Returns:
        root.getNode() (node): Root node of entire FunctionNode representation.
    """
    root = getRoot(lst)
    clauseStack = []
    clauseStack = recursiveConnect(root, clauseStack)
    while (len(clauseStack) != 0):
        c = clauseStack.pop(0)
        connectFromClause(c)
    return root.getNode()


def recursiveConnect(clause, stack):
    """
    Create a stack of clauses, with the leaf clauses at the very top.

    Parameters:
        clause (Clause): Clause object.
        stack (list): Stack of Clause objects.

    Returns:
        new_stack (list): Stack of Clause objects.
    """
    new_stack = stack
    if (clause.isLeaf()):
        new_stack.append(clause)
        return new_stack
    else:
        for child in clause.getChildren():
            new_stack = recursiveConnect(child, new_stack)
        new_stack.append(clause)
    return new_stack

def connectFromClause(clause):
    """
    Create parent/child relationships for the FunctionNodes corresponding to a␣
↪clause and the clause's children.

    Parameters:
        clause (Clause): clause to create relationships for.
```

```
    Returns:
        clause (Clause): clause with created relationships.
    """
    if clause.getNode().returntype == 'CONJ':
        if (clause.leftChild.getNode().returntype != 'PRE-PREDICATE'):
            left_node = LOTlib3.FunctionNode.
↪FunctionNode(returntype='PRE-PREDICATE', name="", parent=clause.getNode(),␣
↪args=[clause.leftChild.getNode()])
            clause.leftChild.getNode().parent = left_node
        else:
            left_node = clause.leftChild.getNode()


        if (clause.rightChild.getNode().returntype != 'CONJ'):
            rNode = LOTlib3.FunctionNode.FunctionNode(returntype='CONJ',␣
↪name='', parent=clause.getNode(), args=[])

            if (clause.rightChild.getNode().returntype != 'PRE-PREDICATE'):
                rNode2 = LOTlib3.FunctionNode.
↪FunctionNode(returntype='PRE-PREDICATE', name="", parent=rNode, args=[clause.
↪rightChild.getNode()])
                clause.rightChild.getNode().parent = rNode2
            else:
                rNode2 = clause.rightChild.getNode()

            rNode.args = [rNode2]
        else:
            rNode = clause.rightChild.getNode()

        clause.getNode().args = [left_node, rNode]

    elif clause.getNode().returntype == 'DISJ':
        if (clause.leftChild.getNode().returntype != 'PRE-PREDICATE'):
            left_node = LOTlib3.FunctionNode.
↪FunctionNode(returntype='PRE-PREDICATE', name="", parent=clause.getNode(),␣
↪args=[clause.leftChild.getNode()])
            clause.leftChild.getNode().parent = left_node
        else:
            left_node = clause.leftChild.getNode()

        if (clause.rightChild.getNode().returntype != 'DISJ'):
            rNode = LOTlib3.FunctionNode.FunctionNode(returntype='DISJ',␣
↪name='', parent=clause.getNode(), args=[])

            if (clause.rightChild.getNode().returntype == 'PREDICATE'):
```

```
                rNode2 = LOTlib3.FunctionNode.
↪FunctionNode(returntype='PRE-PREDICATE', name="", parent=rNode, args=[clause.
↪rightChild.getNode()])
                clause.rightChild.getNode().parent = rNode2
            else:
                rNode2 = clause.rightChild.getNode()

            rNode.args = [rNode2]
        else:
            rNode = clause.rightChild.getNode()

        clause.getNode().args = [left_node, rNode]

    elif clause.getNode().returntype == 'PRE-PREDICATE':
        clause.rightChild.getNode().parent = clause.getNode()
        clause.getNode().args = [clause.rightChild.getNode()]

    return clause


def getRoot(lst):
    """
    Returns the root clause in a list of clauses.

    Parameters:
        lst (list): List of clauses.

    Returns:
        item (Clause): Root clause, with a sentinel parent.
    """
    for item in lst:
        if item.parent.getName() == 'sentinel':
            return item
```

[16]:
```
def convertToNode(string):
    """
    Converts a string into a FunctionNode representation.

    Parameters:
        string (str): String of a lambda function.

    Returns:
        connectTree(l) (FunctionNode): Root node in the FunctionNode␣
↪representation.
        string (str): Original string with exterior parentheses removed.
    """
    string = string[11:-1]
```

```python
    if string[0] == '(':
        t, i = makeClause(string, 1, 1)
        if (i == len(string) - 1):
            string = string[1:-1]
    l = makeClauseList(string, 0)
    recurseClauseList(l)
    return connectTree(l), string

def compareNodeString(node, string):
    """
    Checks whether a FunctionNode object and string representation are␣
 ↪semantically identical.

    Parameters:
        node (FunctionNode): FunctionNode object.
        string (str): String of a lambda function.

    Returns:
        nodeS == stringS (bool): Equality.
    """
    nodeList = re.split('\(|\)', str(node))
    stringList = re.split('\(|\)', string)
    while '' in nodeList:
        nodeList.remove('')
    while '' in stringList:
        stringList.remove('')

    nodeS = ''
    for item in nodeList:
        nodeS += item

    stringS = ''
    for item in stringList:
        stringS += item

    return nodeS == stringS
```

The below code attempts to convert all the Codex strings into FunctionNode objects, and checks whether the newly generated FunctionNode is semantically equivalent to the string representation. If the string is able to be converted accurately, the prior, posterior score, and likelihood of the hypothesis is evaluated on the full set of 9 stimuli. If the string is not converted accurately, an arbitrary value of 88888.88888 is recorded to signify a faulty generated string. Otherwise, if the prior, posterior score, or likelihood cannot be computed, the arbitrary value of 99999.99999 is recorded to signify that.

convertToNode must be enclosed in a try statement as not all of the Codex completions are syntactically correct.

[17]:
```
priors = []
posteriors = []
likelihoods = []
correctResults = []
for i in range(len(data['gen_reshaped'])):
    results = eval(test_prompts['results'][int(i/9)])
    correctResults.append(results)
    try:
        nodeItem, stringItem = convertToNode(data['gen_reshaped'][i])
        assert compareNodeString(nodeItem, stringItem)
    except:
        priors.append(88888.88888)
        posteriors.append(88888.88888)
        likelihoods.append(88888.88888)
        continue

    nodeData = [FunctionData(input=[all_stimuli[i]], output=results[i], alpha=0.
↪999) for i in range(9)]
    newHypothesis = MyHypothesis(value = nodeItem)
    try:
        assert newHypothesis.compute_prior()
        priors.append(newHypothesis.compute_prior())
    except:
        priors.append(99999.99999)

    try:
        assert newHypothesis.compute_posterior(nodeData)
        posteriors.append(newHypothesis.compute_posterior(nodeData))
    except:
        posteriors.append(99999.99999)

    try:
        assert newHypothesis.compute_likelihood(nodeData)
        likelihoods.append(newHypothesis.compute_likelihood(nodeData))
    except:
        likelihoods.append(99999.99999)
```

The resulting dataset is shown below:

[18]:
```
data['priors'] = priors
data['posteriors'] = posteriors
data['likelihoods'] = likelihoods
data['correctResults'] = correctResults
data
```

[18]:
```
      Problem_num  accuracy                              tr_code_concat  \
0               1  0.555556                                        True
```

```
1             1  0.555556                                          True
2             1  1.000000                                          True
3             1  1.000000                                          True
4             1  1.000000                                          True
…           …        …                                           …
1192        133  0.777778  (shape == 'triangle' and color == 'green')
1193        133  0.888889  (shape == 'triangle' and color == 'green')
1194        133  0.888889  (shape == 'triangle' and color == 'green')
1195        133  0.888889  (shape == 'triangle' and color == 'green')
1196        133  0.777778  (shape == 'triangle' and color == 'green')

                          gen_code_concat  true_code_size  gen_code_size  \
0      (color == 'red' or shape == 'circle')               4             37
1      (color == 'red' or shape == 'square')               4             37
2                                      True               4              4
3                                      True               4              4
4                                      True               4              4
…                                        …              …              …
1192 (color == 'blue' and shape == 'square')              42             39
1193                                    False              42              5
1194                                    False              42              5
1195                                    False              42              5
1196                    (shape == 'triangle')              42             21

     num_stims_seen                                      stims_seen  \
0                 1  [{'shape': 'circle', 'color': 'red', 'alpha': …
1                 2  [{'shape': 'circle', 'color': 'red', 'alpha': …
2                 3  [{'shape': 'circle', 'color': 'red', 'alpha': …
3                 4  [{'shape': 'circle', 'color': 'red', 'alpha': …
4                 5  [{'shape': 'circle', 'color': 'red', 'alpha': …
…               …                                                 …
1192              5  [{'shape': 'circle', 'color': 'red', 'alpha': …
1193              6  [{'shape': 'circle', 'color': 'red', 'alpha': …
1194              7  [{'shape': 'circle', 'color': 'red', 'alpha': …
1195              8  [{'shape': 'circle', 'color': 'red', 'alpha': …
1196              9  [{'shape': 'circle', 'color': 'red', 'alpha': …

                                          passed_tests  \
0     ["assert categorize('red', 'circle') == True",…
1     ["assert categorize('red', 'circle') == True",…
2     ["assert categorize('red', 'circle') == True",…
3     ["assert categorize('red', 'circle') == True",…
4     ["assert categorize('red', 'circle') == True",…
…                                                 …
1192  ["assert categorize('red', 'circle') == False"…
1193  ["assert categorize('red', 'circle') == False"…
1194  ["assert categorize('red', 'circle') == False"…
```

```
1195  ["assert categorize('red', 'circle') == False"…
1196  ["assert categorize('red', 'circle') == False"…

                                          failed_tests  \
0     ["assert categorize('blue', 'square') == True"…
1     ["assert categorize('blue', 'circle') == True"…
2                                                   []
3                                                   []
4                                                   []
…                                                  …
1192  ["assert categorize('blue', 'square') == False…
1193  ["assert categorize('green', 'triangle') == Tr…
1194  ["assert categorize('green', 'triangle') == Tr…
1195  ["assert categorize('green', 'triangle') == Tr…
1196  ["assert categorize('red', 'triangle') == Fals…

                                          tr_code_full  \
0                                     "lambda x: True"
1                                     "lambda x: True"
2                                     "lambda x: True"
3                                     "lambda x: True"
4                                     "lambda x: True"
…                                                  …
1192  "lambda x: (x['shape']=='triangle' and x['colo…
1193  "lambda x: (x['shape']=='triangle' and x['colo…
1194  "lambda x: (x['shape']=='triangle' and x['colo…
1195  "lambda x: (x['shape']=='triangle' and x['colo…
1196  "lambda x: (x['shape']=='triangle' and x['colo…

                                          gen_code_full  tr_domain  \
0     def categorize(color, shape):\n\treturn (color…         9
1     def categorize(color, shape):\n\treturn (color…         9
2          def categorize(color, shape):\n\treturn True          9
3          def categorize(color, shape):\n\treturn True          9
4          def categorize(color, shape):\n\treturn True          9
…                                                  …        …
1192  def categorize(color, shape):\n\treturn (color…         1
1193      def categorize(color, shape):\n\treturn False          1
1194      def categorize(color, shape):\n\treturn False          1
1195      def categorize(color, shape):\n\treturn False          1
1196  def categorize(color, shape):\n\treturn (shape…         1

      gen_domain                                     gen_reshaped    priors  \
0              5  "lambda x: (x['color'] == 'red' or x['shape'] … -7.362011
1              5  "lambda x: (x['color'] == 'red' or x['shape'] … -7.362011
2              9                                 "lambda x: True" -1.386294
3              9                                 "lambda x: True" -1.386294
```

```
4             9                                        "lambda x: True" -1.386294
…          …                                                        …      …
1192          1  "lambda x: (x['color'] == 'blue' and x['shape'… -6.620073
1193          0                                       "lambda x: False" -1.386294
1194          0                                       "lambda x: False" -1.386294
1195          0                                       "lambda x: False" -1.386294
1196          3           "lambda x: (x['shape'] == 'triangle')" -1.791759

      posteriors  likelihoods  \
0     -37.768121   -30.406110
1     -37.768121   -30.406110
2      -1.390795    -0.004501
3      -1.390795    -0.004501
4      -1.390795    -0.004501
…          …           …
1192  -52.226988   -45.606915
1193  -54.593612   -53.207317
1194  -54.593612   -53.207317
1195  -54.593612   -53.207317
1196  -32.197870   -30.406110

                                          correctResults
0     [True, True, True, True, True, True, True, Tru…
1     [True, True, True, True, True, True, True, Tru…
2     [True, True, True, True, True, True, True, Tru…
3     [True, True, True, True, True, True, True, Tru…
4     [True, True, True, True, True, True, True, Tru…
…                                                  …
1192  [True, False, True, True, True, True, False, T…
1193  [True, False, True, True, True, True, False, T…
1194  [True, False, True, True, True, True, False, T…
1195  [True, False, True, True, True, True, False, T…
1196  [True, False, True, True, True, True, False, T…

[1197 rows x 19 columns]
```

Next, I analyze the Codex generated functions among the distribution of functions associated with each rule. For each of the 133 rules, for each number of stimuli (1-9), I run the Metropolis Hastings Sampler for 20000 steps and keep track of the top 20 Hypotheses. These Hypotheses are added to the total set of hypotheses corresponding to the rule. After 20 hypotheses are added to the total set at each data amount (1-9), I then add the nine Codex generated hypotheses into the total set.

For each data amount, I compute the posterior for each item in the total set, and use log-sum-exp to normalize the posterior scores.

The runtime for the cell below takes several hours due to Metropolis Hastings, so please skip this cell.

[ ]:
```python
normalized_posteriors = []
for i in range(1, 134):
    print(f"Started rule {i}")
    rule_data = data[data['Problem_num'] == i]
    rule_results = rule_data['correctResults'].iloc[0]
    rule_objects = [FunctionData(input=[all_stimuli[k]],␣
↪output=rule_results[k], alpha=0.999) for k in range(9)]
    top_hypotheses = set()
    codex_hypotheses = set()
    for j in range(1, 10):
        print(f"\tStarted subset {j}")
        codex_hypothesis = rule_data['gen_reshaped'].iloc[j-1]
        sub_objects = rule_objects[:j]
        sub_top = TopN(N=20)
        sub_hypo = MyHypothesis()
        for h in MetropolisHastingsSampler(sub_hypo, sub_objects, steps=20000):
            sub_top << h

        for item in sub_top:
            top_hypotheses.add(item)

        try:
            node, string = convertToNode(codex_hypothesis)
            hy = MyHypothesis(value = node)
            top_hypotheses.add(hy)
            codex_hypotheses.add(hy)
        except:
            continue

    print(f"\tHypothesis set completed.")
    # Get the indices of hypotheses generated by Codex
    top_hypotheses = list(top_hypotheses)
    codexHypoIndices = []
    for item in codex_hypotheses:
        try:
            codexHypoIndices.append(top_hypotheses.index(item))
        except:
            continue

    ## TODO: Evaluate the set top_hypotheses at every individual data amount
    for l in range(1,10):
        postsForAmount = []
        sub_objects = rule_objects[:l]
        for item in top_hypotheses:
            postsForAmount.append(item.compute_posterior(sub_objects))

        postsForAmount_normed = np.array(postsForAmount[:])
```

```
        lse = logsumexp(postsForAmount_normed)
        postsForAmount_normed -= lse
        postsForAmount_normed = np.exp(postsForAmount_normed)

        added = False
        for ind in codexHypoIndices:
            if compareNodeString(str(top_hypotheses[ind]),␣
↪eval(rule_data['gen_reshaped'].iloc[l-1])):
                normalized_posteriors.append(postsForAmount_normed[ind])
                added = True

        if(not added):
            normalized_posteriors.append(0)

normalized_posteriors
data['normalized_posteriors'] = normalized_posteriors
```

Below is an example plotting the normalized posteriors for the total set of Rule 1. We see a Codex prompt with the highest normalized posterior. Checking the index of the Codex prompt, we see it is consistent with how the prompt performs on unit tests (9/9 passed).

[19]:
```
i = 1
rule_data = data[data['Problem_num'] == i]
rule_results = rule_data['correctResults'].iloc[0]
rule_objects = [FunctionData(input=[all_stimuli[k]], output=rule_results[k],␣
↪alpha=0.999) for k in range(9)]
top_hypotheses = set()
codex_hypotheses = set()
for j in range(1, 10):
    codex_hypothesis = rule_data['gen_reshaped'].iloc[j-1]
    sub_objects = rule_objects[:j]
    sub_top = TopN(N=20)
    sub_hypo = MyHypothesis()
    for h in MetropolisHastingsSampler(sub_hypo, sub_objects, steps=1000):
        sub_top << h

    for item in sub_top:
        top_hypotheses.add(item)

    try:
        node, string = convertToNode(codex_hypothesis)
        hy = MyHypothesis(value = node)
        top_hypotheses.add(hy)
        codex_hypotheses.add(hy)
    except:
        continue
```

```
top_hypotheses = list(top_hypotheses)
codexHypoIndices = []
for item in codex_hypotheses:
    print(item)
    correct = 0
    exec(f"def categorize(x): return {str(item)[10:]}")
    for i in range(len(all_stimuli)):
        if categorize(all_stimuli[i]) == rule_results[i]:
            correct += 1
    print(correct)
    try:
        codexHypoIndices.append(top_hypotheses.index(item))
    except:
        print()

codexHypoIndices
```

```
lambda x: True
9
lambda x: (x['color'] == 'red' or x['shape'] == 'circle')
5
lambda x: (x['color'] == 'red' or x['shape'] == 'square')
5
```

[19]:

```
[24, 7, 47]
```

[20]:

```
normalized_posteriors = []
for j in range(1,10):
    postsForAmount = []
    sub_objects = rule_objects[:j]
    for item in top_hypotheses:
        postsForAmount.append(item.compute_posterior(sub_objects))

    postsForAmount_normed = np.array(postsForAmount[:])
    lse = logsumexp(postsForAmount_normed)
    postsForAmount_normed -= lse
    postsForAmount_normed = np.exp(postsForAmount_normed)

    for ind in codexHypoIndices:
        if compareNodeString(str(top_hypotheses[ind]),␣
↪eval(rule_data['gen_reshaped'].iloc[j-1])):
            normalized_posteriors.append(postsForAmount_normed[ind])

posteriorDf = pd.DataFrame()
for j in range(1,10):
    df = pd.DataFrame()
    postsForAmount = []
```

```
    likesForAmount = []
    priorsForAmount = []
    sub_objects = rule_objects[:j]
    for item in top_hypotheses:
        postsForAmount.append(item.compute_posterior(sub_objects))
        likesForAmount.append(item.compute_likelihood(sub_objects))
        priorsForAmount.append(item.compute_prior())

    postsForAmount_normed = np.array(postsForAmount[:])
    lse = logsumexp(postsForAmount_normed)
    postsForAmount_normed -= lse
    postsForAmount_normed = np.exp(postsForAmount_normed)
    df['num_stims'] = [j] * len(postsForAmount)
    hypo_num = list(np.arange(0, len(postsForAmount)))
    for ind in codexHypoIndices:
        hypo_num[ind] = f"Codex Prompt {ind}"

    df['hypo_num'] = hypo_num
    df['scores'] = postsForAmount_normed
    df['unnormed_scores'] = postsForAmount
    df['likelihoods'] = likesForAmount
    df['priors'] = priorsForAmount
    posteriorDf = pd.concat([posteriorDf, df], axis=0)
px.line(posteriorDf, x='num_stims', y='scores', color='hypo_num',␣
 ↪hover_data=['unnormed_scores', 'likelihoods', 'priors'])
```

6 Analysis

Taking the resulting dataset from the previous sections, we now process the data and create visualizations.

[21]:
```
data = pd.read_csv("../data/full_normed.csv").drop('Unnamed: 0', axis=1)
data
```

[21]:
```
      Problem_num  accuracy                              tr_code_concat  \
0               1  0.555556                                        True
1               1  0.555556                                        True
2               1  1.000000                                        True
3               1  1.000000                                        True
4               1  1.000000                                        True
…             …        …                                           …
1192          133  0.777778  (shape == 'triangle' and color == 'green')
1193          133  0.888889  (shape == 'triangle' and color == 'green')
1194          133  0.888889  (shape == 'triangle' and color == 'green')
1195          133  0.888889  (shape == 'triangle' and color == 'green')
1196          133  0.777778  (shape == 'triangle' and color == 'green')
```

```
                         gen_code_concat  true_code_size  gen_code_size  \
0      (color == 'red' or shape == 'circle')               4             37
1      (color == 'red' or shape == 'square')               4             37
2                                     True               4              4
3                                     True               4              4
4                                     True               4              4
…                                        …               …              …
1192  (color == 'blue' and shape == 'square')             42             39
1193                                   False              42              5
1194                                   False              42              5
1195                                   False              42              5
1196                   (shape == 'triangle')              42             21

      num_stims_seen                                        stims_seen  \
0                  1  [{'shape': 'circle', 'color': 'red', 'alpha': …
1                  2  [{'shape': 'circle', 'color': 'red', 'alpha': …
2                  3  [{'shape': 'circle', 'color': 'red', 'alpha': …
3                  4  [{'shape': 'circle', 'color': 'red', 'alpha': …
4                  5  [{'shape': 'circle', 'color': 'red', 'alpha': …
…                  …                                                 …
1192               5  [{'shape': 'circle', 'color': 'red', 'alpha': …
1193               6  [{'shape': 'circle', 'color': 'red', 'alpha': …
1194               7  [{'shape': 'circle', 'color': 'red', 'alpha': …
1195               8  [{'shape': 'circle', 'color': 'red', 'alpha': …
1196               9  [{'shape': 'circle', 'color': 'red', 'alpha': …

                                          passed_tests  \
0     ["assert categorize('red', 'circle') == True",…
1     ["assert categorize('red', 'circle') == True",…
2     ["assert categorize('red', 'circle') == True",…
3     ["assert categorize('red', 'circle') == True",…
4     ["assert categorize('red', 'circle') == True",…
…                                                   …
1192  ["assert categorize('red', 'circle') == False"…
1193  ["assert categorize('red', 'circle') == False"…
1194  ["assert categorize('red', 'circle') == False"…
1195  ["assert categorize('red', 'circle') == False"…
1196  ["assert categorize('red', 'circle') == False"…

                                          failed_tests  \
0     ["assert categorize('blue', 'square') == True"…
1     ["assert categorize('blue', 'circle') == True"…
2                                                   []
3                                                   []
4                                                   []
…                                                   …
1192  ["assert categorize('blue', 'square') == False…
```

```
1193  ["assert categorize('green', 'triangle') == Tr…
1194  ["assert categorize('green', 'triangle') == Tr…
1195  ["assert categorize('green', 'triangle') == Tr…
1196  ["assert categorize('red', 'triangle') == Fals…

                                          tr_code_full  \
0                                     "lambda x: True"
1                                     "lambda x: True"
2                                     "lambda x: True"
3                                     "lambda x: True"
4                                     "lambda x: True"
…                                                  …
1192  "lambda x: (x['shape']=='triangle' and x['colo…
1193  "lambda x: (x['shape']=='triangle' and x['colo…
1194  "lambda x: (x['shape']=='triangle' and x['colo…
1195  "lambda x: (x['shape']=='triangle' and x['colo…
1196  "lambda x: (x['shape']=='triangle' and x['colo…

                                          gen_code_full  tr_domain  \
0     def categorize(color, shape):\n\treturn (color…          9
1     def categorize(color, shape):\n\treturn (color…          9
2          def categorize(color, shape):\n\treturn True          9
3          def categorize(color, shape):\n\treturn True          9
4          def categorize(color, shape):\n\treturn True          9
…                                                  …        …
1192  def categorize(color, shape):\n\treturn (color…          1
1193      def categorize(color, shape):\n\treturn False          1
1194      def categorize(color, shape):\n\treturn False          1
1195      def categorize(color, shape):\n\treturn False          1
1196  def categorize(color, shape):\n\treturn (shape…          1

      gen_domain                                      gen_reshaped    priors  \
0              5  "lambda x: (x['color'] == 'red' or x['shape'] … -7.362011
1              5  "lambda x: (x['color'] == 'red' or x['shape'] … -7.362011
2              9                                  "lambda x: True" -1.386294
3              9                                  "lambda x: True" -1.386294
4              9                                  "lambda x: True" -1.386294
…            …                                                …        …
1192           1  "lambda x: (x['color'] == 'blue' and x['shape'… -6.620073
1193           0                                 "lambda x: False" -1.386294
1194           0                                 "lambda x: False" -1.386294
1195           0                                 "lambda x: False" -1.386294
1196           3             "lambda x: (x['shape'] == 'triangle')" -1.791759

      posteriors  likelihoods  \
0     -37.768121   -30.406110
1     -37.768121   -30.406110
```

```
2      -1.390795    -0.004501
3      -1.390795    -0.004501
4      -1.390795    -0.004501
…          …            …
1192  -52.226988   -45.606915
1193  -54.593612   -53.207317
1194  -54.593612   -53.207317
1195  -54.593612   -53.207317
1196  -32.197870   -30.406110

                                          correctResults  normalized_posteriors
0     [True, True, True, True, True, True, True, Tru…               0.001422
1     [True, True, True, True, True, True, True, Tru…               0.001654
2     [True, True, True, True, True, True, True, Tru…               0.710069
3     [True, True, True, True, True, True, True, Tru…               0.868221
4     [True, True, True, True, True, True, True, Tru…               0.870282
…                                                  …                      …
1192  [True, False, True, True, True, True, False, T…               0.000002
1193  [True, False, True, True, True, True, False, T…               0.874313
1194  [True, False, True, True, True, True, False, T…               0.985238
1195  [True, False, True, True, True, True, False, T…               0.985301
1196  [True, False, True, True, True, True, False, T…               0.000098

[1197 rows x 20 columns]
```

Below are some jitter functions to make scatter plots more clear.

[22]:
```python
def rand_jitter(arr):
    stdev = .03 * (max(arr) - min(arr))
    return arr + np.random.randn(len(arr)) * stdev

def packaged_jitter(df, x, y, c):
    return jitter(df[x], df[y], c=df[c])

def jitter(x, y, s=20, c='b', marker='o', cmap=None, norm=None, vmin=None,␣
 ↪vmax=None, alpha=None, linewidths=None, verts=None, hold=None, **kwargs):
    return plt.scatter(rand_jitter(x), rand_jitter(y), s=s, c=c, marker=marker,␣
 ↪cmap=cmap, norm=norm, vmin=vmin, vmax=vmax, alpha=alpha,␣
 ↪linewidths=linewidths, **kwargs)
```

Creating tr_priors, the priors of the actual code; actual_domains, the number of stimuli predicted to be True by the actual code; generated_domains, the number of stimuli predicted to be True by the Codex completion; Type, the classification of a concept by counting the presence of conjunctions and disjunctions in the actual code; tr_complexity, number of clauses present in actual code; gen_complexity, number of clauses present in Codex completion.

[23]:

```
tr_priors = []
for i in range(len(data['tr_code_full'])):
    results = eval(test_prompts['results'][int(i/9)])
    nodeItem, stringItem = convertToNode(data['tr_code_full'][i].replace('==',␣
 ↪' == '))
    newHypothesis = MyHypothesis(value=nodeItem)
    tr_priors.append(newHypothesis.compute_prior())

data['tr_priors'] = tr_priors
```

[24]:

```
actual_domains = []
generated_domains = []
for i in range(len(data['failed_tests'])):
    act_domain = 0
    gen_domain = 0
    failed_set = eval(data['failed_tests'][i])
    for item in failed_set:
        bool_val = item[len(item) - 5:].replace(" True", "True")
        if (eval(bool_val)):
            act_domain += 1
        else:
            gen_domain += 1

    passed_set = eval(data['passed_tests'][i])
    for item in passed_set:
        bool_val = item[len(item) - 5:].replace(" True", "True")
        if (eval(bool_val)):
            act_domain += 1
            gen_domain += 1

    assert act_domain <= 9
    assert gen_domain <= 9
    actual_domains.append(act_domain)
    generated_domains.append(gen_domain)

assert len(actual_domains) == len(generated_domains)
assert len(actual_domains) == len(data['failed_tests'])

data['tr_domain'] = actual_domains
data['gen_domain'] = generated_domains
```

[25]:

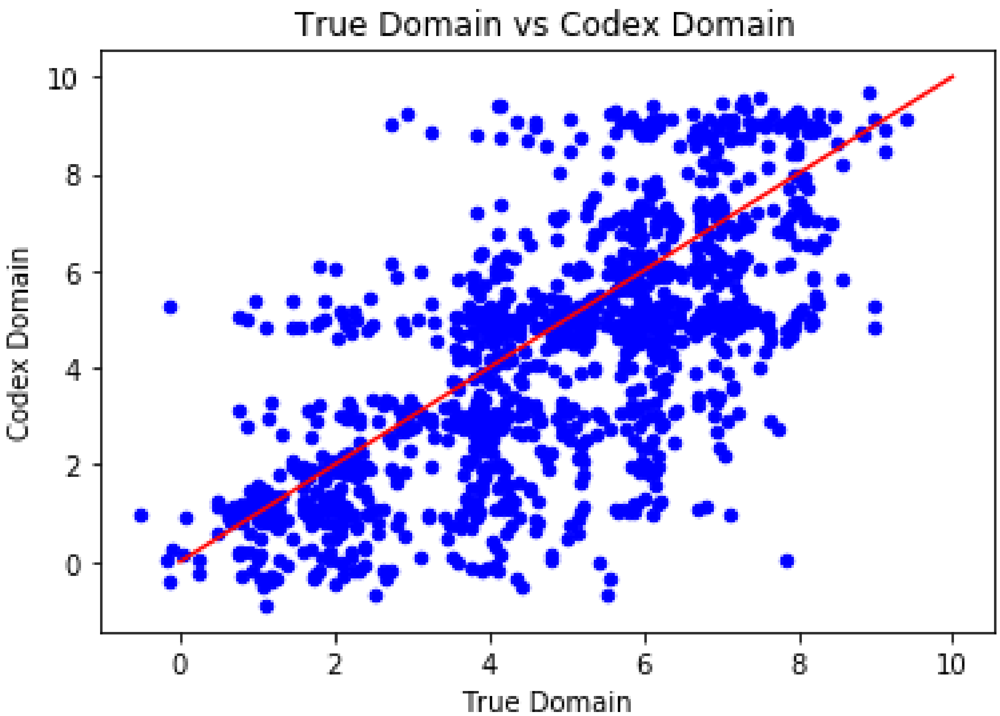
```
nornand = []
for i in data['tr_code_concat']:
    nor = i.count('or') - i.count('color')
    nand = i.count('and')
    if (nor-nand < 0):
        nornand.append('conjunction')
```

```
    elif (nor-nand > 0):
        nornand.append('disjunction')
    else:
        nornand.append('both')
data['Type'] = nornand

data_conjunc = data[data['Type'] == 'conjunction']
```

[26]:
```
tr_complexity = []
for i in data['tr_code_concat']:
    it = i.replace('color', 'colar')
    ct = it.count('colar == ') + it.count('shape ==') + it.count('and') + it.
 ↪count('or') + it.count('not') + it.count('True') + it.count('False')
    tr_complexity.append(ct)
data['tr_complexity'] = tr_complexity

gen_complexity = []
for i in data['gen_code_concat']:
    it = i.replace('color', 'colar')
    ct = it.count('colar == ') + it.count('shape ==') + it.count('and') + it.
 ↪count('or') + it.count('not') + it.count('True') + it.count('False')
    gen_complexity.append(ct)
data['gen_complexity'] = gen_complexity
```

Some analysis of the domains (stimuli predicted to be True according to learned concept) between the actual (base code generated by LOTlib3) code and generated code.

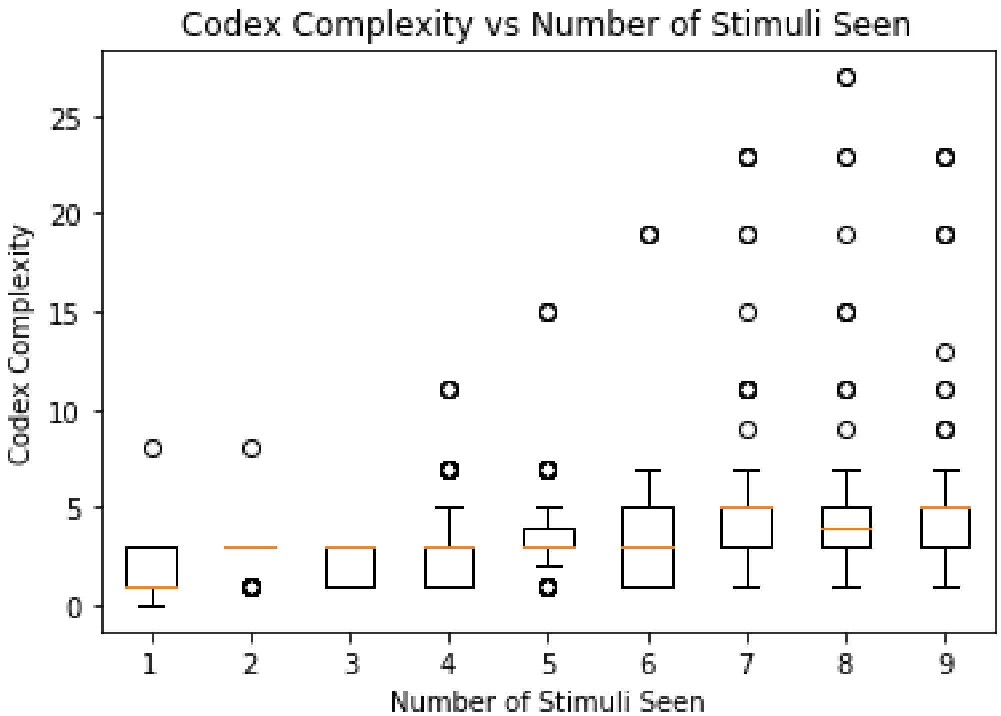
[27]:
```
jitter(data['tr_domain'], data['gen_domain'])
plt.plot(range(11), range(11), color='r')
plt.xlabel("True Domain")
plt.title("True Domain vs Codex Domain")
plt.ylabel("Codex Domain");
```



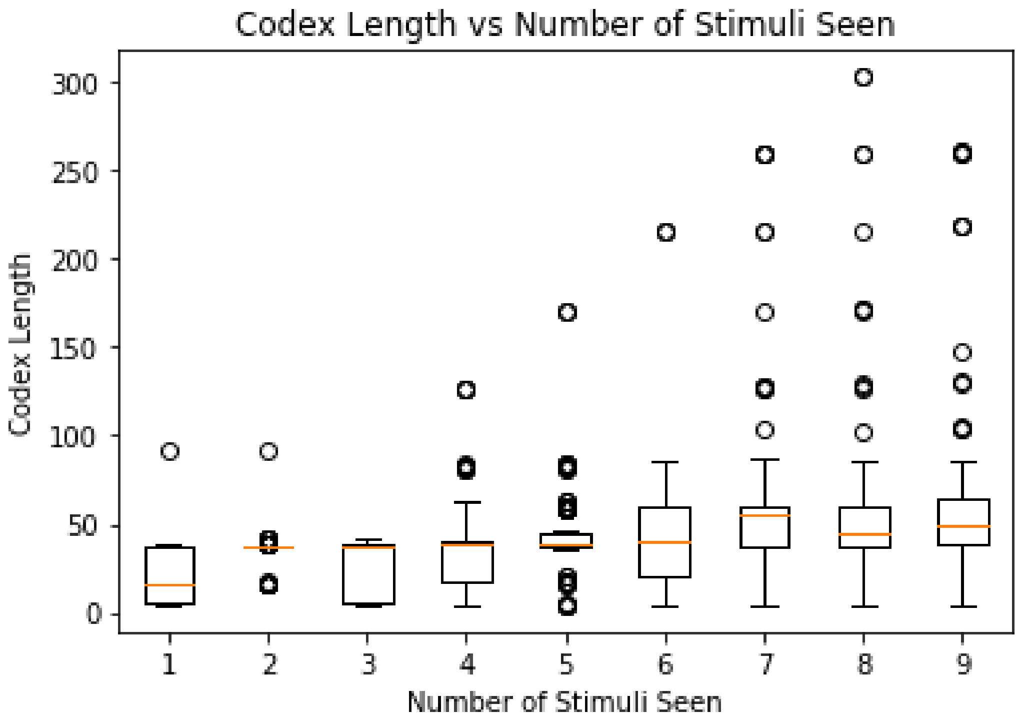
Below are three different plots comparing the complexity of Codex generated functions to the complexity of the ground truth. The comparisons are done in three different ways and metrics: 1) number of clauses (predicates or connectives), 2) length of the function string, and 3) computed prior.

Notice there is a steeper increase in median measures between 1 stimuli and 2 stimuli for both the complexity and length plots. Additionally, there is a steep decrease in the prior between 1 stimuli and 2 stimuli for the priors plot. This supports the idea of Codex having a bias towards simplicity, which is a phenomenon also seen in human subjects.

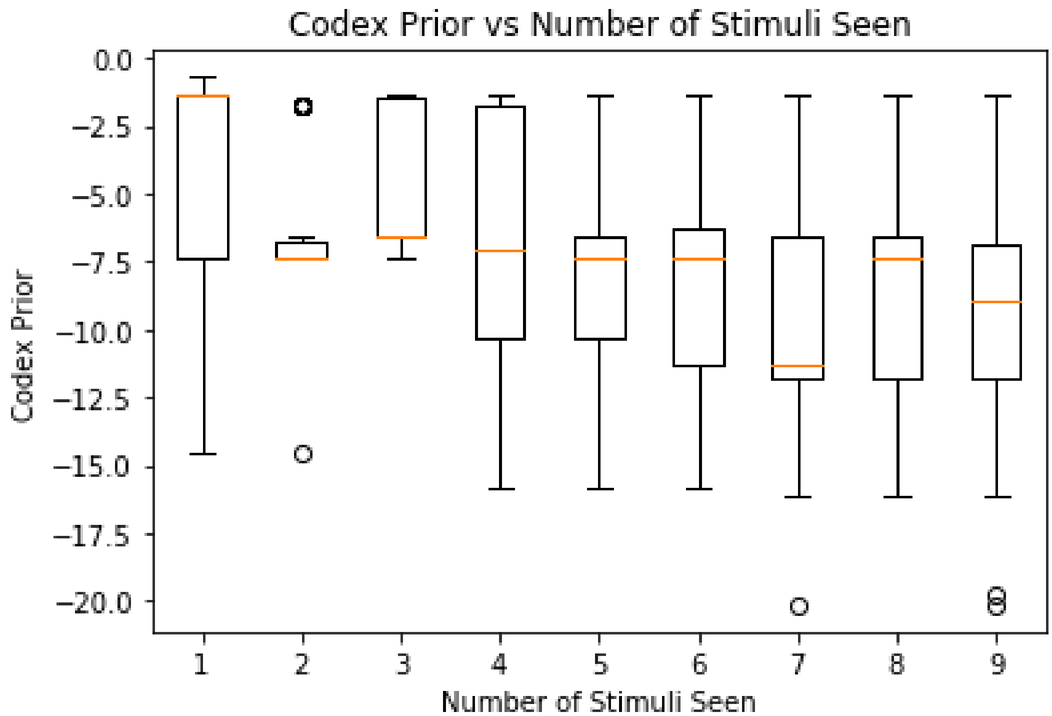
[28]:
```
paccs = pd.DataFrame()
for i in range(1, 10):
    paccs[f"{i}"] = data[data['num_stims_seen'] == i].
 ↪reset_index()['gen_complexity']

plt.boxplot(paccs.swapaxes("index", "columns"))
plt.title('Codex Complexity vs Number of Stimuli Seen')
plt.xlabel('Number of Stimuli Seen')
plt.ylabel('Codex Complexity');
```



[29]:
```
paccs = pd.DataFrame()
for i in range(1, 10):
    paccs[f"{i}"] = data[data['num_stims_seen'] == i].
 ↪reset_index()['gen_code_size']

plt.boxplot(paccs.swapaxes("index", "columns"))
plt.title('Codex Length vs Number of Stimuli Seen')
plt.xlabel('Number of Stimuli Seen')
plt.ylabel('Codex Length');
```

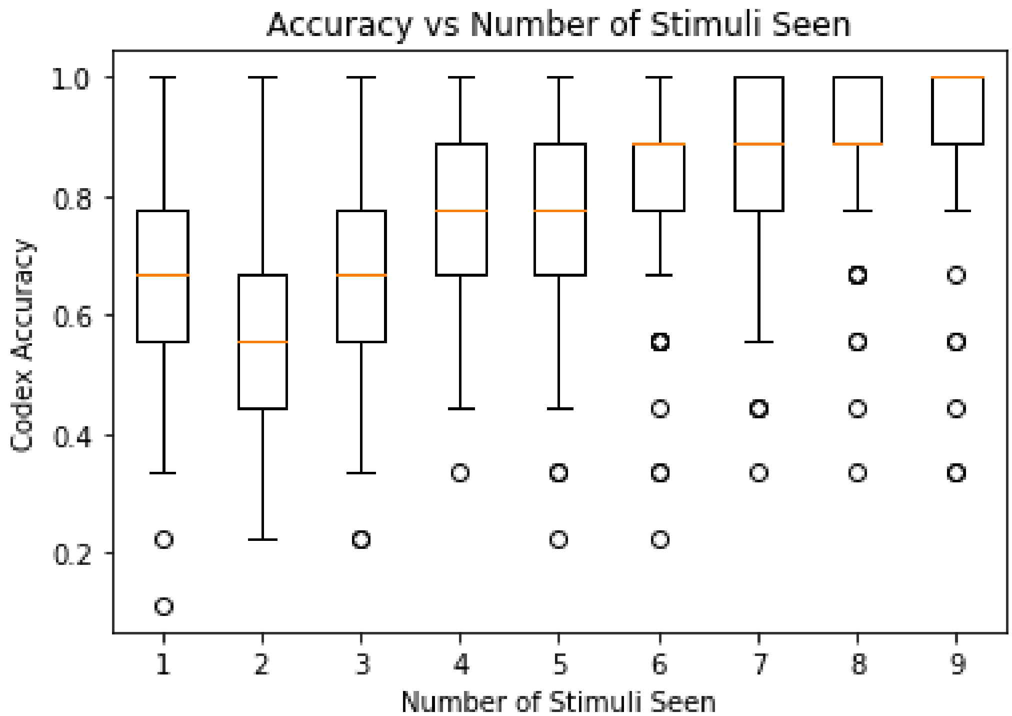


[30]:
```
paccs = pd.DataFrame()
for i in range(1, 10):
    d = data[(data['priors'] != 88888.88888) & (data['priors'] != 99999.99999)]
    paccs[f"{i} num"] = d[d['num_stims_seen'] == i].reset_index()['priors']

plt.boxplot(paccs.swapaxes("index", "columns"))
plt.title('Codex Prior vs Number of Stimuli Seen')
plt.xlabel('Number of Stimuli Seen')
plt.ylabel('Codex Prior')
plt.show();
```



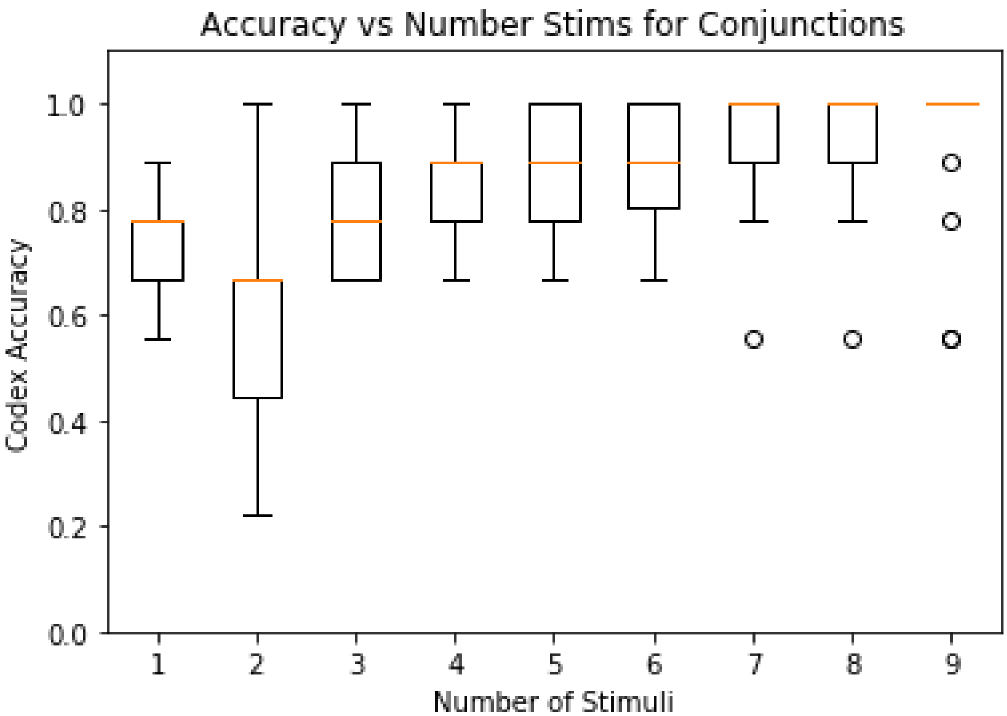
Next is a boxplot of accuracies across the number of stimuli seen by Codex. Notice an upward trajectory as the number of stimuli seen increases, which indicates that Codex is successfully learning from new examples.

[31]:
```
paccs = pd.DataFrame()
for i in range(1, 10):
    paccs[f"{i}"] = data[data['num_stims_seen'] == i].reset_index()['accuracy']
plt.boxplot(paccs.swapaxes("index", "columns"))
plt.title('Accuracy vs Number of Stimuli Seen')
plt.xlabel('Number of Stimuli Seen')
plt.ylabel('Codex Accuracy');
```

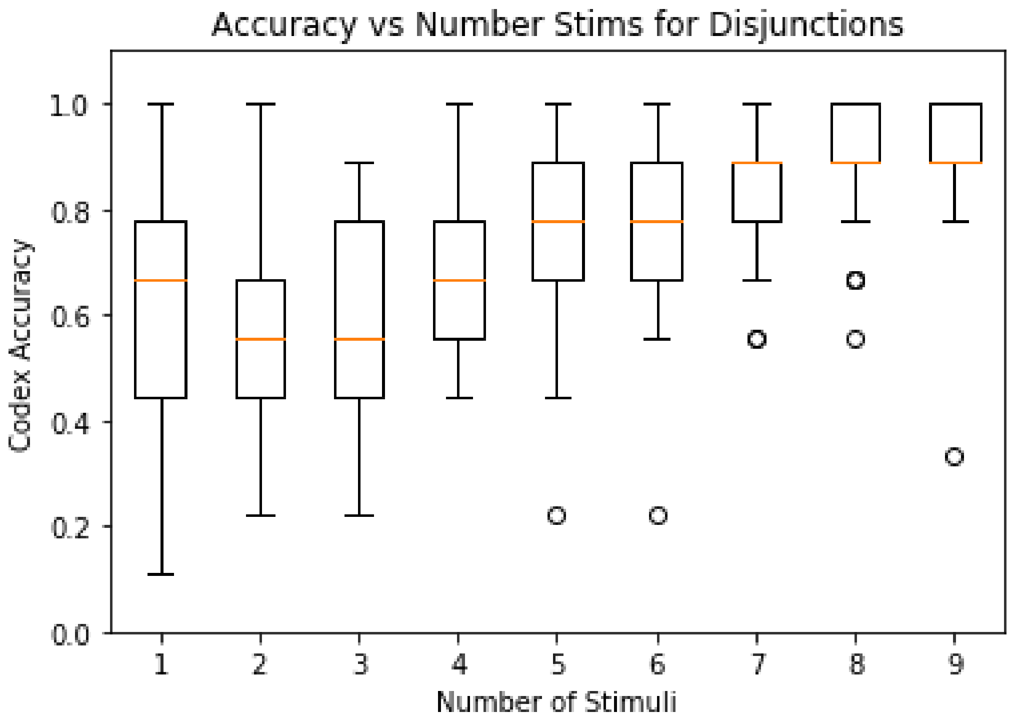


The next two graphs plot the accuracies of Codex across number of stimuli seen for conjunction and disjunction rules separately. Notice the median accuracies for conjunction rules are higher than for disjunction rules, which supports the and/or asymmetry found in Shepard et al. 1961.

[32]:
```
paccs = pd.DataFrame()
for i in range(1, 10):
    paccs[f"{i}"] = data_conjunc[data_conjunc['num_stims_seen'] == i].
 ↪reset_index()['accuracy']
plt.boxplot(paccs.swapaxes("index", "columns"))
plt.title('Accuracy vs Number Stims for Conjunctions')
plt.xlabel("Number of Stimuli")
plt.ylabel("Codex Accuracy")
plt.ylim((0, 1.1));
```

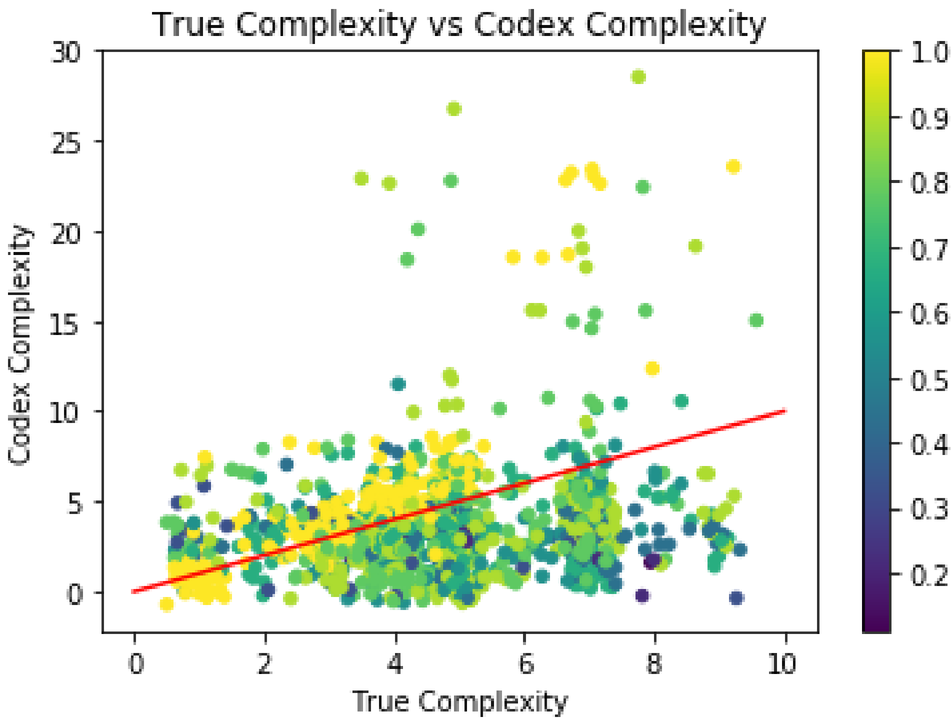



```
[33]: data_disjunc = data[data['Type'] == 'disjunction']
paccs = pd.DataFrame()
for i in range(1, 10):
    paccs[f"{i}"] = data_disjunc[data_disjunc['num_stims_seen'] == i].
 ↪reset_index()['accuracy']
plt.boxplot(paccs.swapaxes("index", "columns"));
plt.title('Accuracy vs Number Stims for Disjunctions')
plt.xlabel('Number of Stimuli')
plt.ylabel('Codex Accuracy')
plt.ylim((0, 1.1));
```



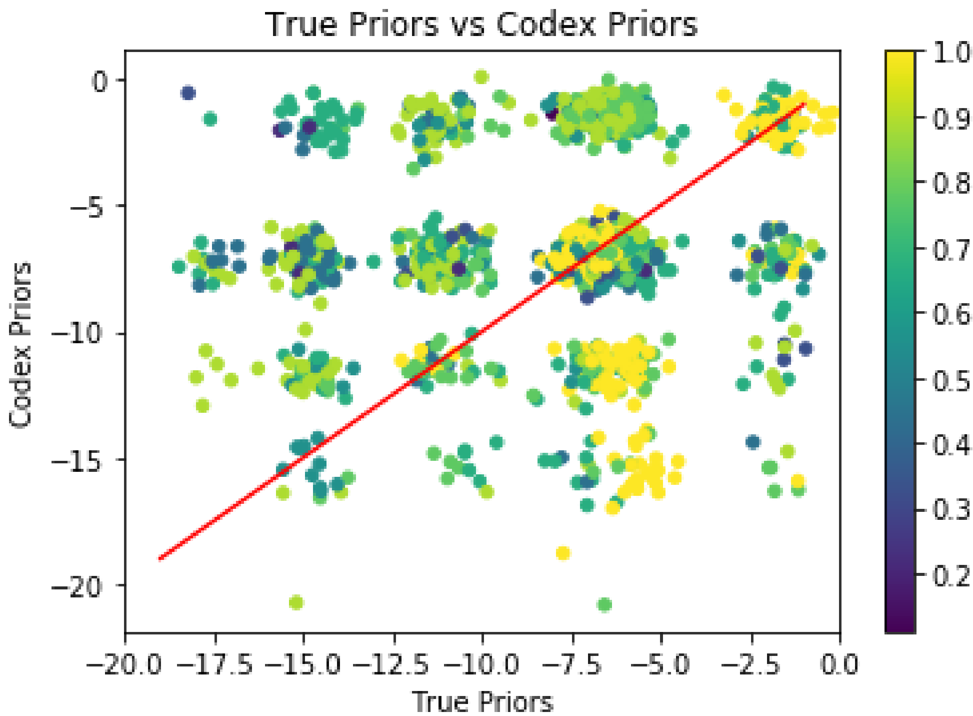
The next plot scatters complexity of the actual code against complexity of the Codex code. Notice that accuracies are in general higher when the two complexities are similar (near the red y=x line). Also note that for higher true complexities, the generated code tends to become much more complex than the actual code. This indicates the tendency for Codex to shift from generalization to memorization at higher complexities.

[34]:
```
packaged_jitter(data, 'tr_complexity', 'gen_complexity', 'accuracy')
plt.plot(range(11), range(11), color='r')
plt.xlabel('True Complexity')
plt.ylabel('Codex Complexity')
plt.title('True Complexity vs Codex Complexity')
plt.colorbar();
```



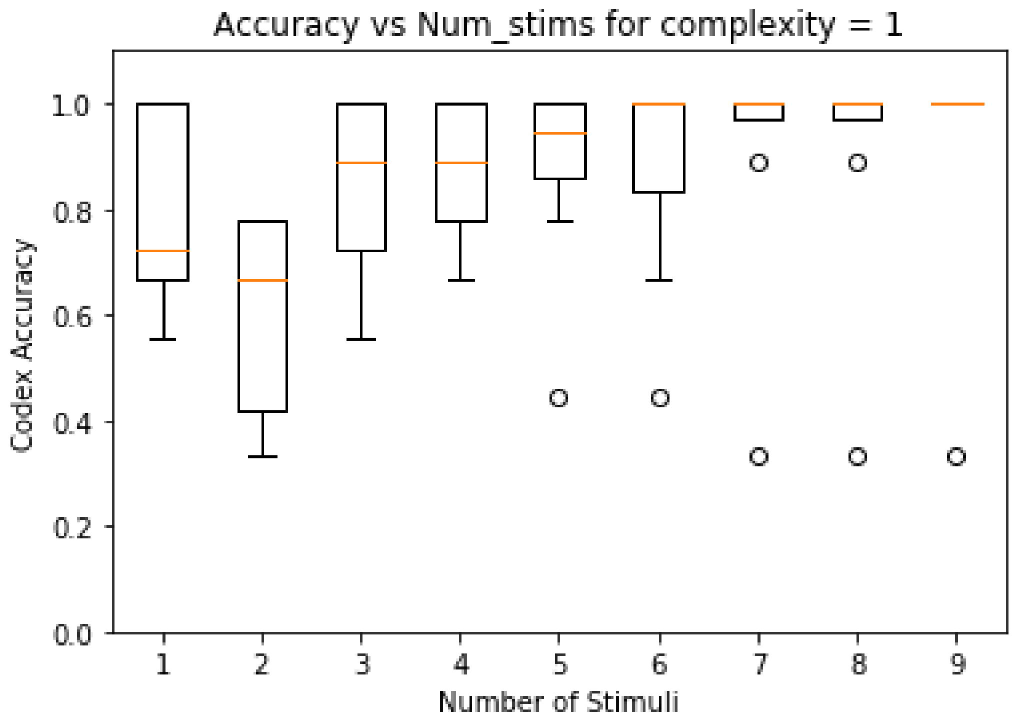
Next, a scatterplot of the actual code priors and generated code priors. We see higher accuracies when the priors are similar, as well as when the true priors are larger than the generated priors. The lowest accuracies converge in the region where the generated priors are larger than the true priors.

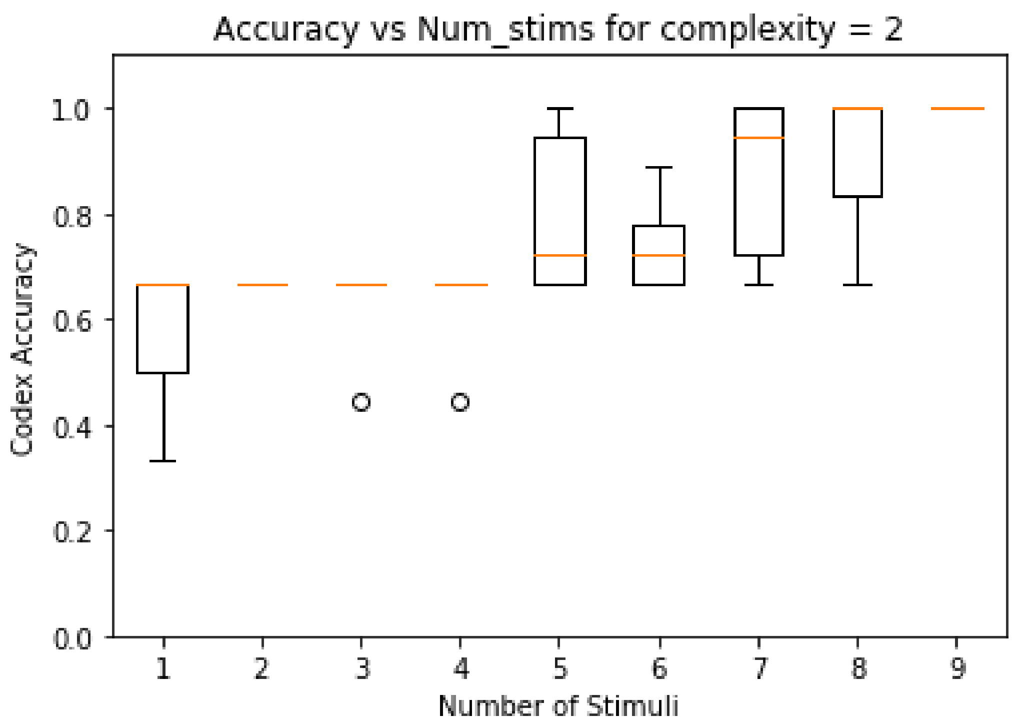
[35]:
```
clean = data[(data['priors'] != 88888.88888) & (data['priors'] != 99999.99999)␣
 ↪& (data['priors'] != float("-inf"))]
packaged_jitter(clean, "tr_priors", "priors", "accuracy")
plt.colorbar()
plt.xlim((-20, 0))
plt.plot(range(-19, 0), range(-19, 0), color='r')
plt.xlabel('True Priors')
plt.title("True Priors vs Codex Priors")
plt.ylabel('Codex Priors');
```

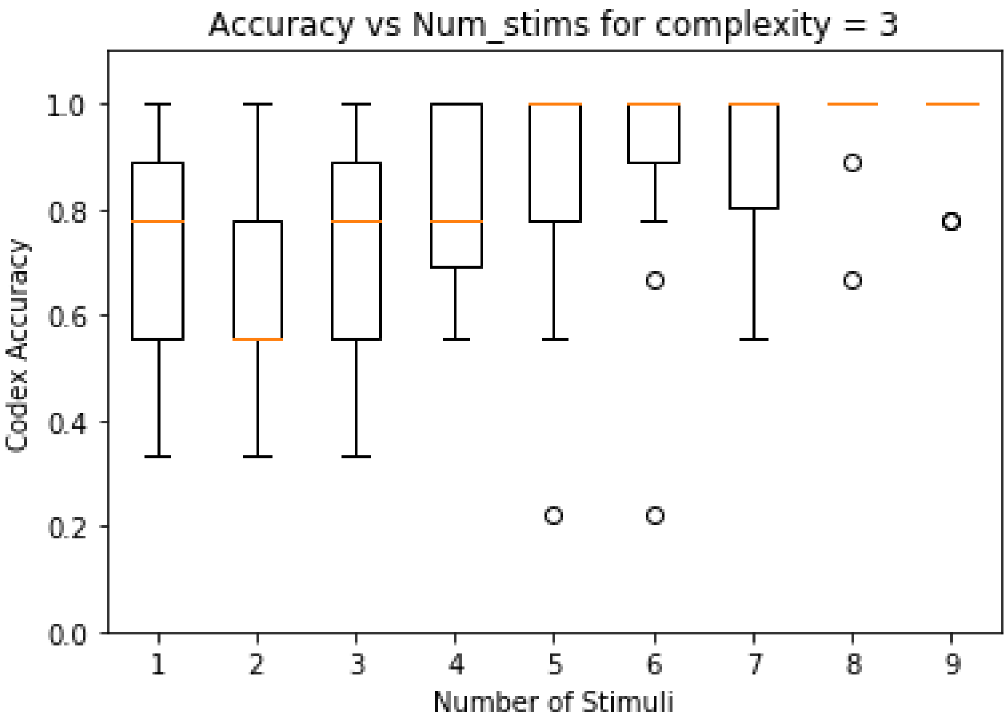


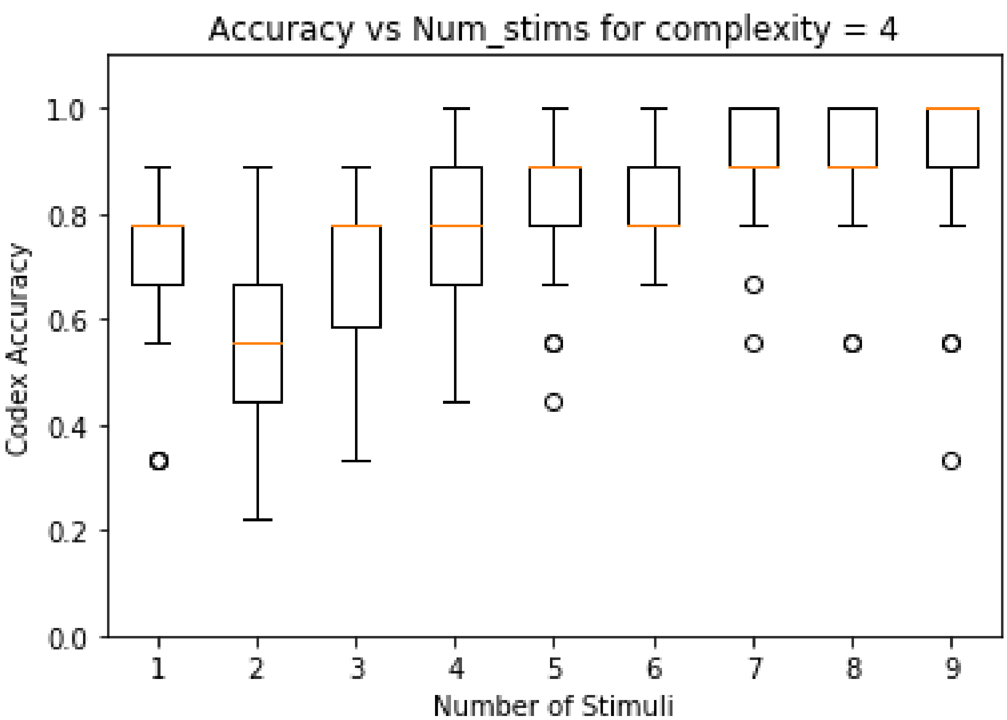
The below cell confirms some basic intuitions about code complexity. Notice the median accuracies for Codex are higher for lower true complexity. The median accuracies steadily decreases as the complexity of the true code increases.

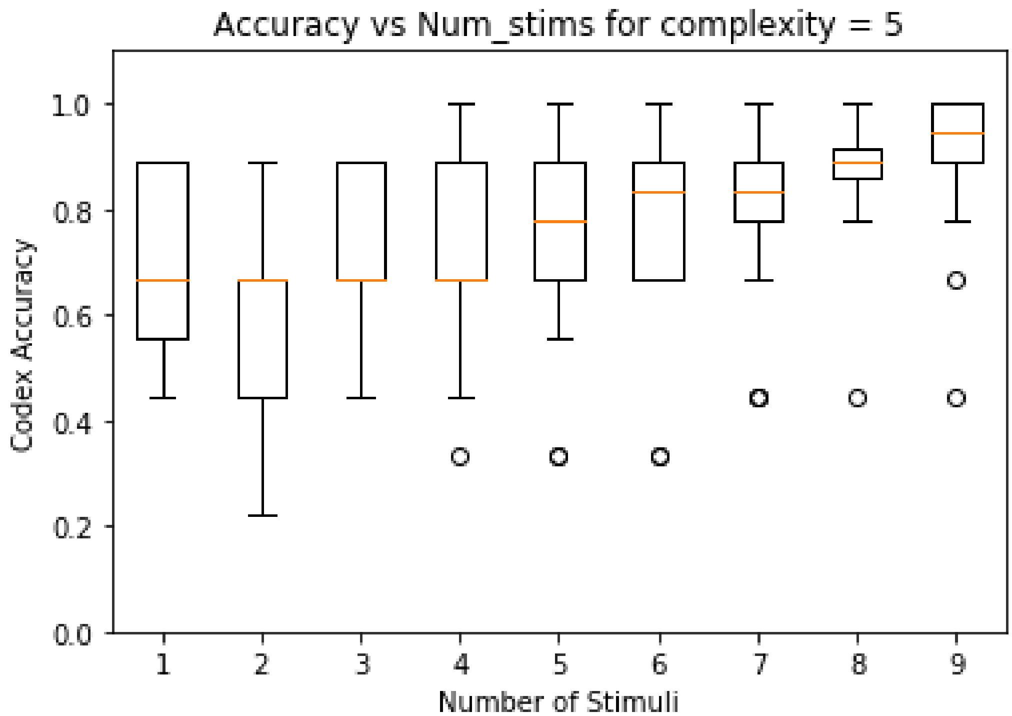
[36]:
```
for j in range(1, max(data['tr_complexity']) + 1):
    exec(f"fig{j} = plt.figure('Figure {j}')")
    data_com = data[data['tr_complexity'] == j]
    paccs = pd.DataFrame()
    for i in range(1, 10):
        paccs[f"{i}"] = data_com[data_com['num_stims_seen'] == i].
↪reset_index()['accuracy']
    plt.boxplot(paccs.swapaxes("index", "columns"))
    plt.title(f"Accuracy vs Num_stims for complexity = {j}")
    plt.xlabel('Number of Stimuli')
    plt.ylabel('Codex Accuracy')
    plt.ylim((0, 1.1));
```

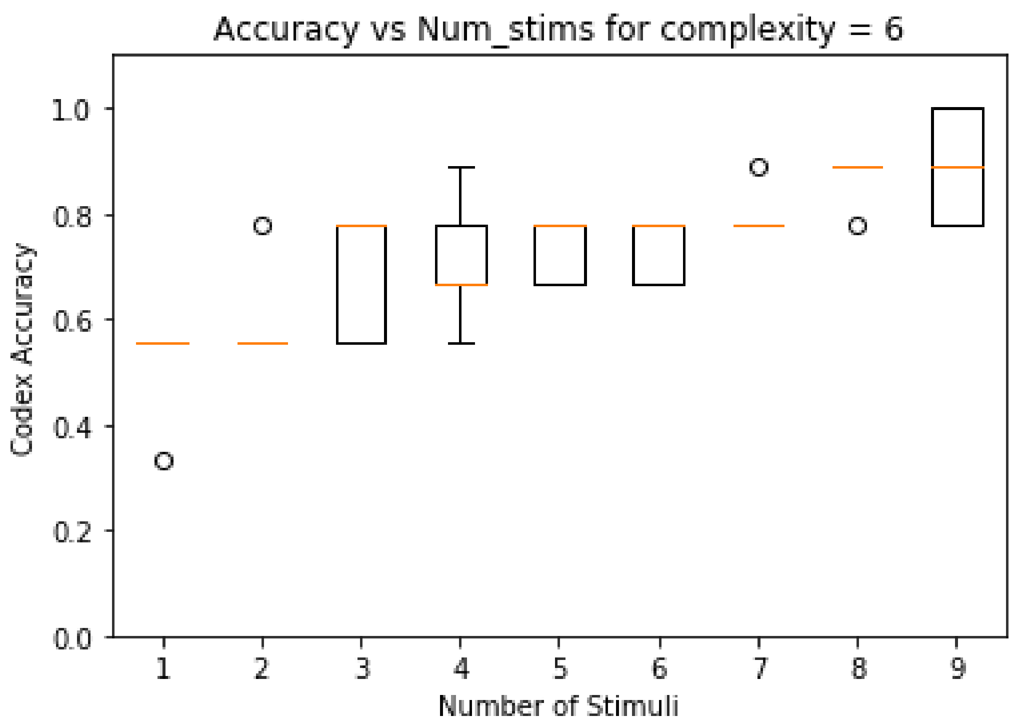

Accuracy vs Num_stims for complexity = 1
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9

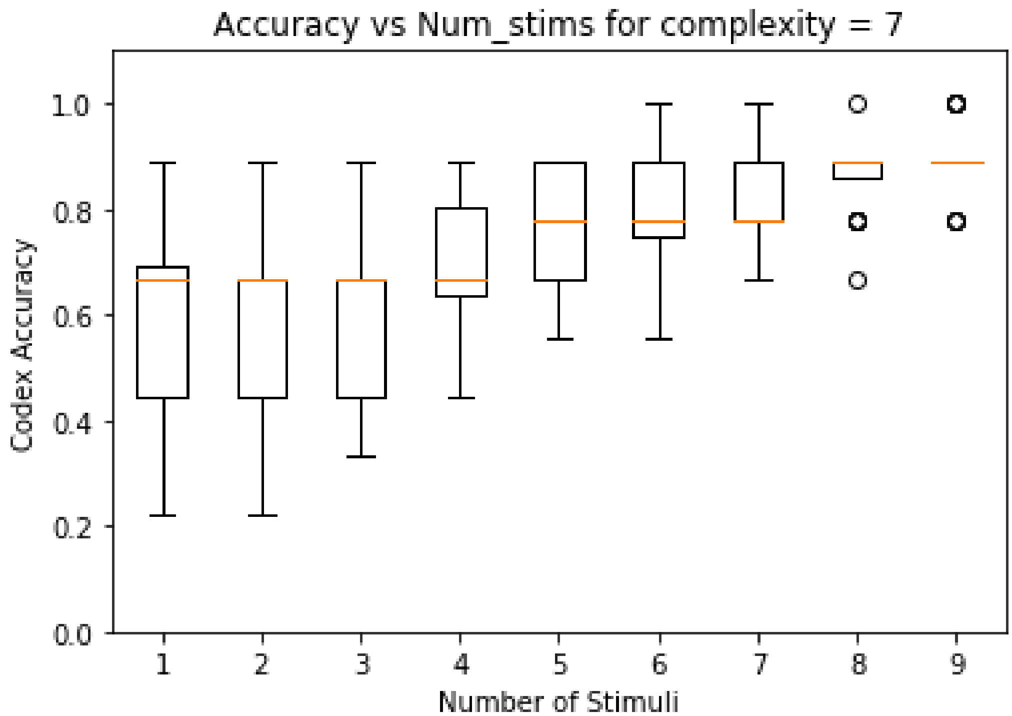

Accuracy vs Num_stims for complexity = 2
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9

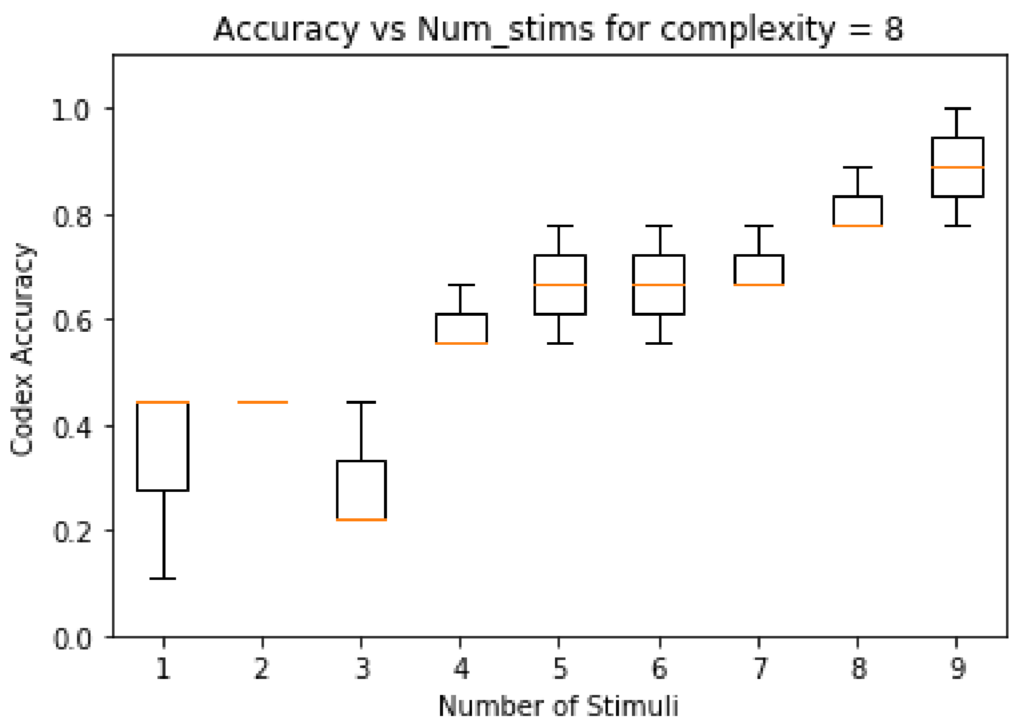

Accuracy vs Num_stims for complexity = 3
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9

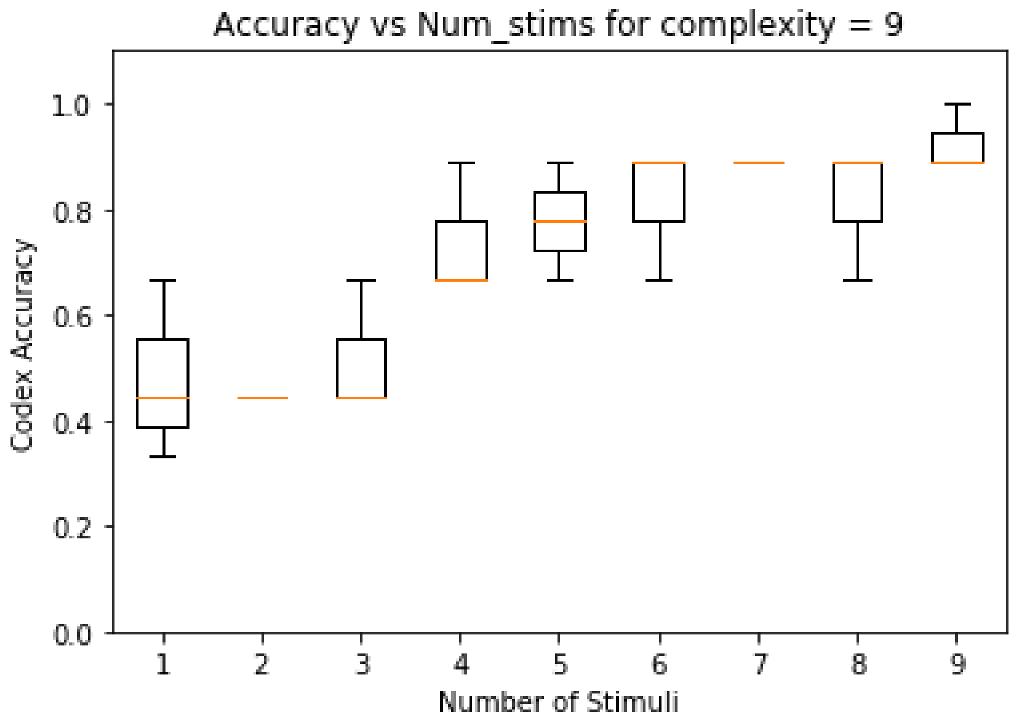

Accuracy vs Num_stims for complexity = 4
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9


Accuracy vs Num_stims for complexity = 5
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9


Accuracy vs Num_stims for complexity = 6
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9

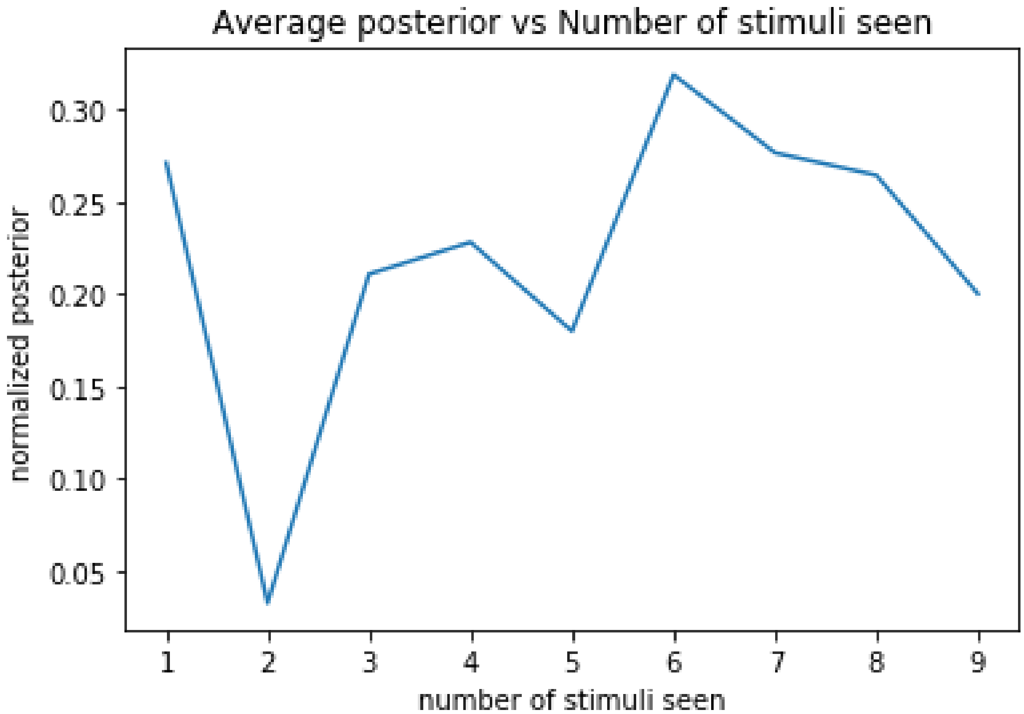

Accuracy vs Num_stims for complexity = 7
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9


Accuracy vs Num_stims for complexity = 8
Codex Accuracy
Number of Stimuli
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6
7
8
9



Below is a plot of the average normalized posterior for Codex across the number of seen stimuli. I am calculating normalized posterior for each rule at that data amount, which means that dips in the plot may be because the Codex generated code before the dip and after the dip are different.

[37]:
```
d = data[(data['likelihoods'] != 88888.88888) & (data['likelihoods'] != 99999.
↪99999) &
     (data['priors'] != 88888.88888) & (data['priors'] != 99999.99999) &
     (data['posteriors'] != 88888.88888) & (data['posteriors'] != 99999.99999)]
d = d.groupby('num_stims_seen').mean()
plt.plot(d['normalized_posteriors'])
plt.title('Average posterior vs Number of stimuli seen')
plt.xlabel('number of stimuli seen')
plt.ylabel('normalized posterior');
```



7 Conclusion

In summary, the experimentation on Codex shows promising hints that neural networks share similar reasoning biases as humans. Going forward, we plan to expand experimentation to more complex rule domains and tasks, an example of which is to introduce quantifiers to the rule grammar. I also plan to compare the prompt engineering approach against the fine-tuning approach, as well as replicate results with other language models, such as CodeT5, CodeBERT, and BLOOM. These branches will allow us to tap into larger pools of literature, in an effort to explain why neural networks are demonstrating these biases.

I would like to thank my faculty mentors, Professor Knight and Professor Piantadosi, as well as my graduate student mentor Mark. I also extensively used the data and tools in the Knight Lab and Colala, for which I am very grateful. And last but not least, I would like to thank the SURF program for a very educational summer.