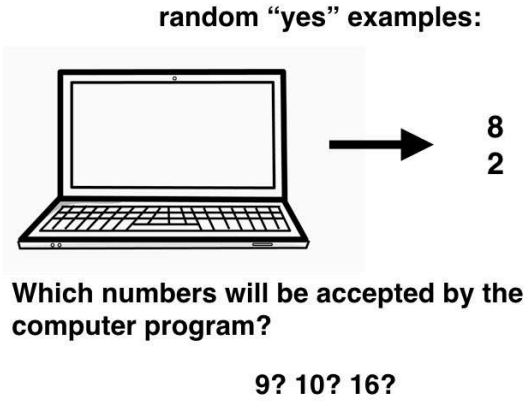
Homework - Bayesian modeling - Part A (100 points)

Bayesian concept learning with the number game

by *Brenden Lake* and *Todd Gureckis*
Computational Cognitive Modeling
NYU class webpage: https://brendenlake.github.io/CCM-site/

In this notebook, you will implement (mostly from scratch!) a Bayesian concept learning model and the "number game," as covered in lecture. As with so many of our everyday inferences, the data we receive is far too sparse and noisy to be conclusive. Nevertheless, people must make generalizations and take actions based on imperfect and insufficient data. In data science and machine learning, the situation is often the same: the data is not enough to produce an answer with certainty, yet we can make meaningful generalizations anyway. What computational mechanisms can support these types of inferences?

The number game is a quintessential inductive problem. In the number game, there is an unknown computer program that generates numbers in the range 1 to 100. You are provided with a small set of random examples from this program. For instance, in the figure below, you get two random examples from the program: the numbers '8' and '2'.



Which numbers will also be accepted by the same program? Of course, it depends what the program is, and you don't have enough information to be sure. Should '9' be accepted? Perhaps, if the concept is "all numbers up to 10." What about '10'? A better candidate, since the program could again be "numbers up to 10", or "all even numbers." What about '16'? This is another good candidate, and the program "powers of 2" is also consistent with the examples so far. How should one generalize based on the evidence so far? This homework explores how the Bayesian framework provides an answer to this question.

You should read the following paper carefully.

Josh Tenenbaum's paper introduced the number game. You can download the paper on EdStem:

- Tenenbaum, J. B. (2000). Rules and similarity in concept learning. In Advances in Neural Information Processing Systems (NIPS).

The Bayesian model

In the number game, we receive a set of n positive examples $X = \{x^{(1)}, \ldots, x^{(n)}\}$ of an unknown concept C. In a Bayesian analysis of the task, the goal is predict $P(y \in C \mid X)$, which is the probability that a new number y is also a member of the concept C after receiving the set of examples X.

Updating beliefs with Bayes' rule

Let's proceed with the Bayesian model of the task. There is a hypothesis space H of concepts, where a particular member of the hypothesis space (i.e., a particular concept) is denoted $h \in H$. The Bayesian model includes a prior distribution $P(h)$ over the hypotheses and a likelihood $P(X|h)$. Bayes' rule specifies how to compute the posterior distribution over hypotheses given these two pieces:

$$P(h|X) = \frac{P(X|h)P(h)}{\sum_{h' \in H} P(X|h')P(h')} \tag{1}$$

The likelihood and prior are specified below.

Likelihood

We assume that each number in X is an independent sample from the set of all valid numbers. Thus, the likelihood decomposes as a product of individual probabilities,

$$P(X|h) = \prod_{i=1}^{n} P(x^{(i)}|h). \tag{2}$$

We assume that the numbers are sampled uniformly at random from the set of valid numbers, such that $P(x^{(i)}|h) = \frac{1}{|h|}$ if $x^{(i)} \in h$ and $P(x^{(i)}|h) = 0$ otherwise. The term $|h|$ is the cardinality or set size of the hypothesis h.

Prior

The hypothesis space H includes two main kinds of hypotheses. You can think of each hypothesis as a list of the numbers that fit that hypothesis.

- The first kind consists of mathematical hypotheses such as odd numbers, even numbers, square numbers, cube numbers, primes, multiples of n,

powers of n, and numbers ending with a particular digit. Each mathematical hypothesis is given equal weight in the prior.

- The second kind consists of interval hypotheses, which are solid intervals of numbers, such as $12, 13, 14, 15, 16, 17$. Intervals of intermediate size are favored (rather than very small or large hypotheses) by reweighting according to an Erlang distribution, $P(h) \propto \frac{|h|}{\sigma^2}\exp -|h|/\sigma$ such that $\sigma = 10$.

There is a free parameters `mylambda` controls how much of the prior is specified by each type of hypothesis, with `mylambda` weight going to the mathematical hypotheses and `1-mylambda` weights going to the interval hypotheses.

We provide starter code below that generates the mathematical hypotheses and their prior probabilities (in natural log space).

Making Bayesian predictions

Once we have the posterior beliefs over hypotheses, we want to be able to make predictions about the membership of a new number y in the concept C, or as mentioned $P(y \in C \mid X)$. To compute this, we average over all possible hypotheses weighted by the posterior probability,

$$P(y \in C \mid X) = \sum_{h \in H} P(y \in C \mid h) P(h|X), \tag{3}$$

where the first term is simply 1 or 0 based on the membership of y in h, and the second term is the posterior weight.

In [2]:
```
# Here are some packaegs that may be useful
from __future__ import print_function
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import logsumexp
from tqdm import tqdm
x_max = 100 # (numbers between 1 and 100 are allowed)
```

In [3]:
```
# Generate a list of all mathematical hypotheses
def make_h_odd():
    return list(range(1,x_max+1,2))

def make_h_even():
    return list(range(2,x_max+1,2))

def make_h_square():
    h = []
    for x in range(1,x_max+1):
        if x**2 <= x_max:
            h.append(x**2)
    return h

def make_h_cube():
    h = []
    for x in range(1,x_max+1):
```

```
        if x**3 <= x_max:
            h.append(x**3)
    return h

def make_h_primes():
    return [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,

def make_h_mult_of_y(y):
    h = []
    for x in range(1,x_max+1):
        if x*y <= x_max:
            h.append(x*y)
    return h

def make_h_powers_of_y(y):
    h = []
    for x in range(1,x_max+1):
        if y**x <= x_max:
            h.append(y**x)
    return h

def make_h_numbers_ending_in_y(y):
    h = []
    for x in range(1,x_max+1):
        if str(x)[-1] == str(y):
            h.append(x)
    return h

def generate_math_hypotheses(mylambda):
    h_set = [make_h_odd(), make_h_even(), make_h_square(), make_h_cube(), make_h
    h_set += [make_h_mult_of_y(y) for y in range(3,13)]
    h_set += [make_h_powers_of_y(y) for y in range(2,11)]
    h_set += [make_h_numbers_ending_in_y(y) for y in range(0,10)]
    n_hyp = len(h_set)
    log_prior = np.log(mylambda * np.ones(n_hyp) / float(n_hyp))
    return h_set, log_prior

h_set_math, log_prior_math = generate_math_hypotheses(2./3)
print("Four examples of math hypotheses:")
for i in range(4):
    print(h_set_math[i])
    print("")
print("Their prior log-probabilities:")
print(log_prior_math[0:4])
```

```
Four examples of math hypotheses:
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 4
3, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 8
3, 85, 87, 89, 91, 93, 95, 97, 99]

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 8
4, 86, 88, 90, 92, 94, 96, 98, 100]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

[1, 8, 27, 64]

Their prior log-probabilities:
[-3.93182563 -3.93182563 -3.93182563 -3.93182563]
```

In [4]:
```
## Generate a list of all interval hypotheses
def make_h_between_y_and_z(y,z):
    assert(y >= 1 and z <= x_max)
    return list(range(y,z+1))

def pdf_erlang(x,sigma=10.):
    return (x / sigma**2) * np.exp(-x/sigma)

def generate_interval_hypotheses(mylambda):
    h_set = []
    for y in range(1,x_max+1):
        for z in range(y,x_max+1):
            h_set.append(make_h_between_y_and_z(y,z))
    nh = len(h_set)
    pv = np.ones(nh)
    for idx,h in enumerate(h_set): # prior based on length
        pv[idx] = pdf_erlang(len(h))
    pv = pv / np.sum(pv)
    pv = (1-mylambda) * pv
    log_prior = np.log(pv)
    return h_set, log_prior

h_set_int, log_prior_int = generate_interval_hypotheses(2./3)
print("Four examples of interval hypotheses")
for i in range(4):
    print(h_set_int[i])
    print("")
print("Their prior log-probabilities:")
print(log_prior_int[0:4])
```

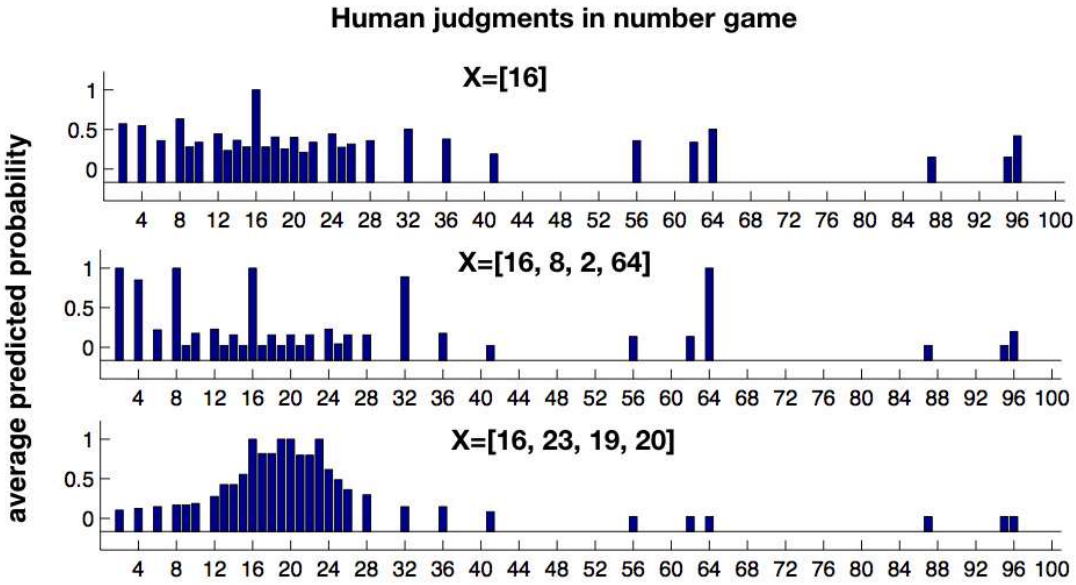
```
Four examples of interval hypotheses
[1]

[1, 2]

[1, 2, 3]

[1, 2, 3, 4]

Their prior log-probabilities:
[-10.197254    -9.60410682  -9.29864171  -9.11095964]
```

Human behavioral judgments

Tenenbaum ran eight participants in an experiment where they were provided with various sets X of random positive examples from a concept. They were asked to rate the probability that each of 30 test numbers would belong to the same concept of the observed examples.

The following plot shows the mean rating across the human participants for three different sets. Note that since only 30 test numbers were evaluated, and thus a

value of 0 in the plot indicates missing data (rather than zero probability).



Your goal is to implement the Bayesian concept learning model in order to produce the same plots, although with the model judgements rather than the human judgements.

In [5]:
```
# The 30 test numbers that Tenenbaum used are here
x_eval = [2,4,6,8,9,10]+list(range(12,23))+[24,25,26,28,32,36,41,56,62,64,87,95,
```

Implementation

Problem 1 (90 points)

Your main task is to produce the same three plots as shown above, although showing model predictions rather than human behavioral judgements. To do so, you'll need to implement the Bayesian concept learning model. A successful implementation will include the following components:

- A function for computing the log-probability of a hypothesis h according to the prior (largely provided in starter code).
- A function for computing the log-likelihood of a set of numbers X given a particular hypothesis h.
- A function for computing the log-posterior over all hypotheses h given a set of numbers X that were sampled from h.
- According to the "Making Bayesian predictions" section above, a function for computing the probability that a new number y belongs to the same concept as a set of sampled numbers X
- Code for making the plots

Tip: For probabilistic modeling in general, we like to compute probabilities in log-space to help avoid numerical issues such as underflow. For instance, rather than

multiplying the prior and likelihood (resulting in potentially very small numbers), we sum the log-prior and the log-likelihood. Also, check out the nifty `logsumexp` function (see scipy doc) which is used to normalize log-probability distributions in a numerically safer way. This function is already loaded.

In [6]:
```
# Your implementation goes here
# init the data
h_set = h_set_math + h_set_int

X1 = [16]
X2 = [16, 8, 2, 64]
X3 = [16, 23, 19, 20]

# A function for computing the log-probability of a hypothesis h according to th
def get_log_prob_h(h_idx):
    if h_idx < len(h_set_math):
        return log_prior_math[h_idx]
    else:
        return log_prior_int[h_idx-len(h_set_math)]
```

In [7]:
```
# A function for computing the log-likelihood of a set of numbers X given a part
def get_log_likelihood_X_given_h(X, h):
    if set(X).issubset(set(h)):
        return len(X)*np.log(1/len(h))
    else:
        return np.log(0)
```

In [8]:
```
# A function for computing the log-posterior over all hypotheses h given a set o
log_prob_X1_insert_h_arr = np.zeros(len(h_set))
log_prob_X2_insert_h_arr = np.zeros_like(log_prob_X1_insert_h_arr)
log_prob_X3_insert_h_arr = np.zeros_like(log_prob_X1_insert_h_arr)

for h_idx, h in enumerate(h_set):
    log_prob_X1_insert_h_arr[h_idx] = get_log_likelihood_X_given_h(X1, h)+get_lo
    log_prob_X2_insert_h_arr[h_idx] = get_log_likelihood_X_given_h(X2, h)+get_lo
    log_prob_X3_insert_h_arr[h_idx] = get_log_likelihood_X_given_h(X3, h)+get_lo

log_posterior_h_given_X1 = log_prob_X1_insert_h_arr - logsumexp(log_prob_X1_inse
log_posterior_h_given_X2 = log_prob_X2_insert_h_arr - logsumexp(log_prob_X2_inse
log_posterior_h_given_X3 = log_prob_X3_insert_h_arr - logsumexp(log_prob_X3_inse

def get_log_posterior_h_given_X(h_idx, X):
    if X == X1:
        return log_posterior_h_given_X1[h_idx]
    elif X == X2:
        return log_posterior_h_given_X2[h_idx]
    elif X == X3:
        return log_posterior_h_given_X3[h_idx]
```

```
C:\Users\liaof\AppData\Local\Temp\ipykernel_26408\3843687945.py:6: RuntimeWarnin
g: divide by zero encountered in log
  return np.log(0)
```

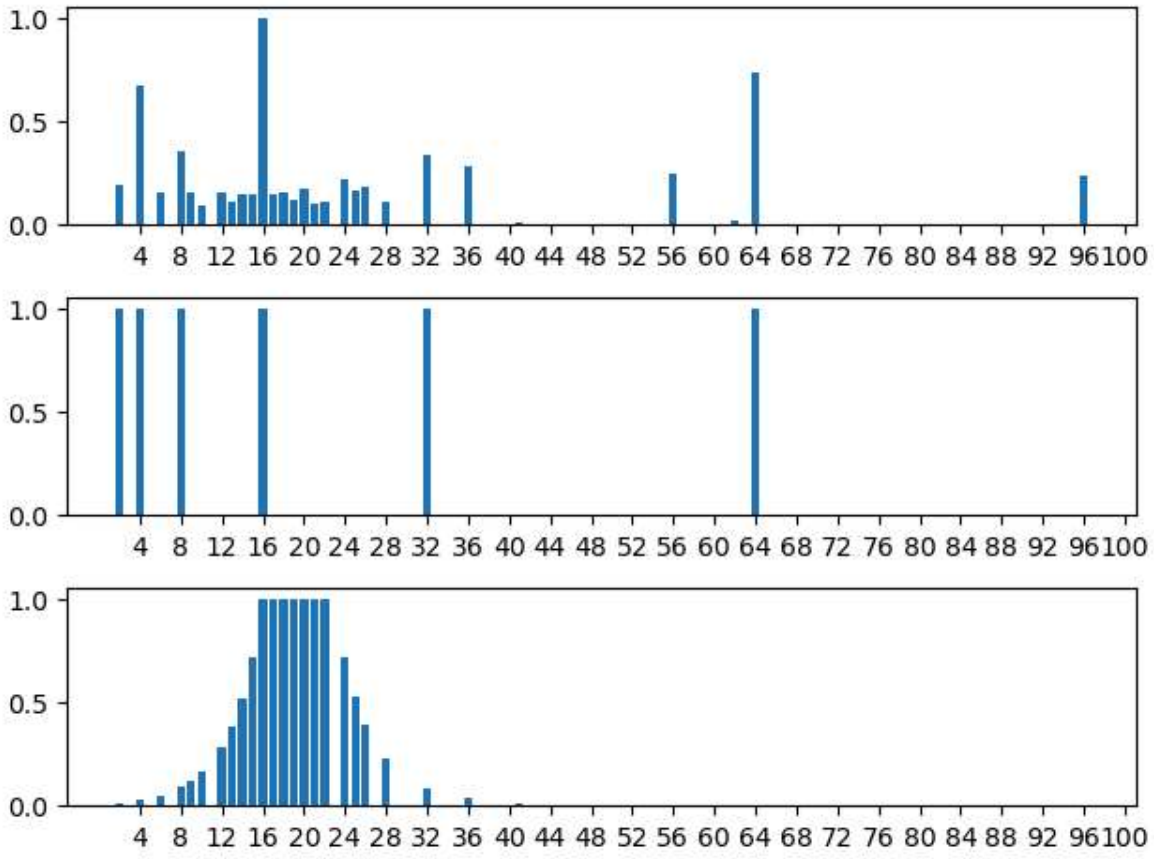
In [9]:
```
# A function for computing the probability that a new number y belongs to the sa
def get_prob_y_in_C_given_X(y, X):
    log_posterior_h_given_X_list = []

    for h_idx in range(len(h_set)):
```

```
        h = h_set[h_idx]
        if y in h:
            log_posterior_h_given_X_list.append(get_log_posterior_h_given_X(h_id
        else:
            log_posterior_h_given_X_list.append(np.log(0))

    prob_y_in_C_given_X = np.exp(logsumexp(log_posterior_h_given_X_list))
    return prob_y_in_C_given_X

# print(get_prob_y_in_C_given_X(16, X1))
```

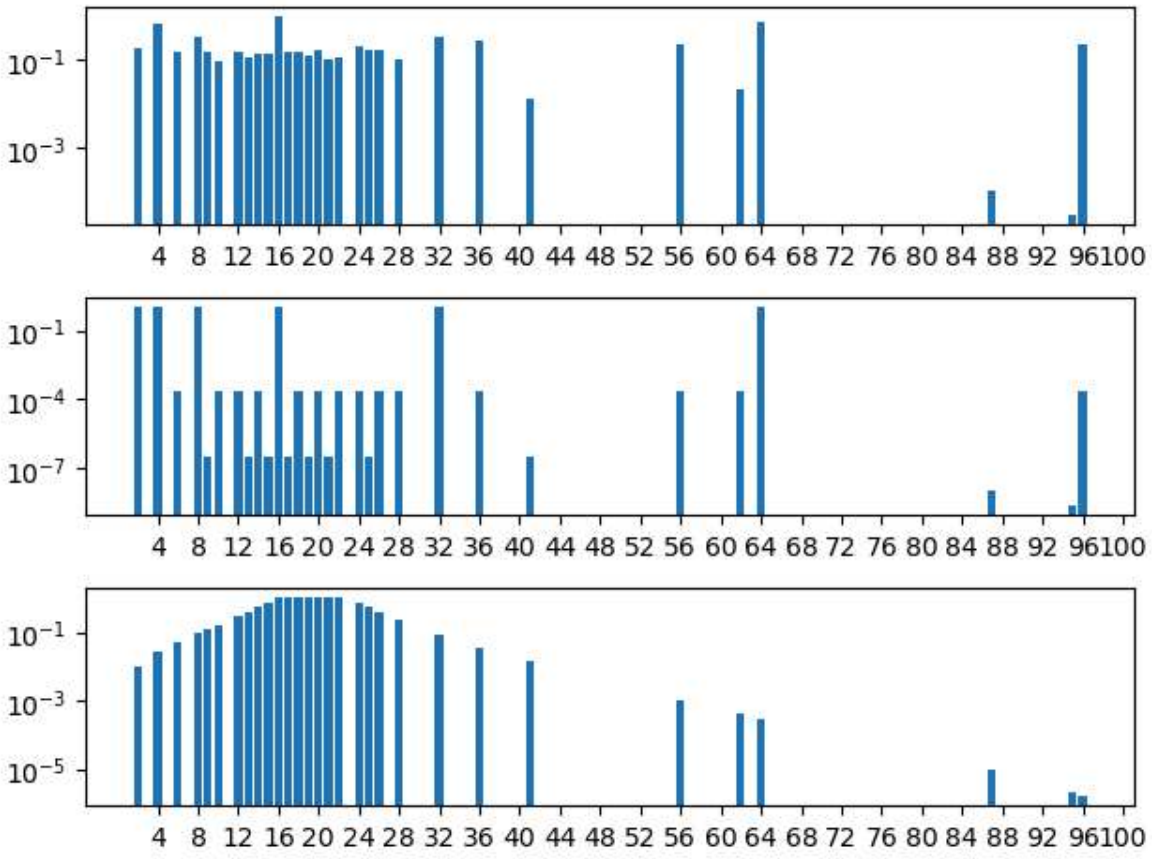
In [10]:

```
# Code for making the plots
prob_y_in_C_given_X1_list = []
prob_y_in_C_given_X2_list = []
prob_y_in_C_given_X3_list = []

x_ticks = range(4, 100+4, 4)

for y in x_eval:
    prob_y_in_C_given_X1_list.append(get_prob_y_in_C_given_X(y, X1))
    prob_y_in_C_given_X2_list.append(get_prob_y_in_C_given_X(y, X2))
    prob_y_in_C_given_X3_list.append(get_prob_y_in_C_given_X(y, X3))

plt.figure()
plt.subplot(311)
plt.bar(x_eval, prob_y_in_C_given_X1_list)
plt.xticks(x_ticks)
plt.subplot(312)
plt.bar(x_eval, prob_y_in_C_given_X2_list)
plt.xticks(x_ticks)
plt.subplot(313)
plt.bar(x_eval, prob_y_in_C_given_X3_list)
plt.xticks(x_ticks)

plt.tight_layout()
plt.show()
```

```
C:\Users\liaof\AppData\Local\Temp\ipykernel_26408\424716102.py:10: RuntimeWarnin
g: divide by zero encountered in log
  log_posterior_h_given_X_list.append(np.log(0))
```



In [13]:
```
# Since some probablities are too small, use log
plt.figure()

plt.subplot(311)
plt.bar(x_eval, prob_y_in_C_given_X1_list)
plt.xticks(x_ticks)
plt.yscale('log')

plt.subplot(312)
plt.bar(x_eval, prob_y_in_C_given_X2_list)
plt.xticks(x_ticks)
plt.yscale('log')

plt.subplot(313)
plt.bar(x_eval, prob_y_in_C_given_X3_list)
plt.xticks(x_ticks)
plt.yscale('log')

plt.tight_layout()
plt.show()
```



Problem 2 (10 points)

Discuss your general thoughts on this Bayesian model to understand human judgments in the number game. Discussion questions could include the following (as well as others):

- Is the model convincing? Why or why not?
- Is the number game and Bayesian model relevant to more naturalistic settings for concept learning in childhood or everyday life?
- Where could the hypothesis space come from?
- What algorithms could people be using to approximate Bayesian inference, rather than enumerating all the hypotheses, as in the current implementation?

Please write a short response in the cell below. Your response should be about two paragraphs.

YOUR RESPONSE GOES HERE

1.Is the model convincing? Why or why not?

Yes, Bayesian model is quite convincing, since it indeed provide reasonable predictions that closed to human's (According to the plots above, they have the same peaks and similar trend). And its arithmetic principle is quite close to human

brains, which is find the maximum conditional probability based on limited information.

2.Is the number game and Bayesian model relevant to more naturalistic settings for concept learning in childhood or everyday life?

Yes, maybe. As far as I am concerned, children know quite little about the world, and the way parents help them to understand classifications by making examples is quite similar to the way we use example data X to help Bayesian model learn what number might be in the set. They all use limited information to get the conditional probability.

3.Where could the hypothesis space come from?

In this case, our Bayesian model's hypothesis space come from common number sets that follow some rules or similarities, like our math sets and integer sets. In real life, hypothesis space might come from researchers' domain knowledge, work and life experience ect.

4.What algorithms could people be using to approximate Bayesian inference, rather than enumerating all the hypotheses, as in the current implementation?

People can use Markov Chain Monte Carlo (MCMC) algorithms such as Metropolis-Hastings method to approximate Bayesian inference, which can be quite useful if we don't want to find all the hypothese and find its posterior probability.