Homework - Bayesian modeling - Part A (95 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/ (https://brendenlake.github.io/CCM-site/)

This homework is due before midnight on Monday, April 4.

In this notebook, you will get hands on experience with Bayesian concept learning 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.

The number game was introduced in the following paper:

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

This is assignment is adapted from exercises developed by Josh Tenenbaum.

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:

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

The likelihood is specified below. We will leave the prior to later.

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,

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

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.

Problem 1 (10 points)

Let's compute a very simple posterior distribution by hand. Assume there are only two possible hypotheses:

- multiples of 10 (h_1)
- even numbers (h_2)

which are equally likely in the prior. Only the numbers 1 through 100 are possible. The positive examples X consist of just 10 and 30. What is the posterior probability of each hypothesis? Please **show your work** in the cell below.

Please double check that your equations display properly when printing your homework. We cannot give you credit if your equations are missing. It's your responsibility to check.

YOUR ANSWER GOES HERE

Given,

$$P(h|X) = \frac{P(X|h)P(h)}{\sum_{h' \in H} P(X|h')P(h')}$$
$$P(h_1) = 1/2$$
$$P(h_2) = 1/2$$

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

where $P(x^{(i)}|h) = \frac{1}{|h|}$ 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.

Then,

$$P(X|h_1) = \prod_{i=1}^{n} P(x^{(i)}|h_1) = (\frac{1}{10})(\frac{1}{10}) = \frac{1}{100}$$
$$P(X|h_2) = \prod_{i=1}^{n} P(x^{(i)}|h_2) = (\frac{1}{50})(\frac{1}{50}) = \frac{1}{2500}$$
$$\sum_{h' \in H} P(X|h')P(h') = P(X|h_1)P(h_1) + P(X|h_2)P(h_2) = (\frac{1}{10})(\frac{1}{10})(\frac{1}{2}) + (\frac{1}{50})(\frac{1}{50})(\frac{1}{2}) = \frac{13}{2500}$$

Finally,

$$P(h_1|X) = \frac{P(X|h_1)P(h_1)}{\sum_{h' \in H} P(X|h')P(h')} = \frac{(\frac{1}{100})(\frac{1}{2})}{(\frac{13}{2500})} = \frac{25}{26} \approx 0.9615$$
$$P(h_2|X) = \frac{P(X|h_2)P(h_2)}{\sum_{h' \in H} P(X|h')P(h')} = \frac{(\frac{1}{2500})(\frac{1}{2})}{(\frac{13}{2500})} = \frac{1}{26} \approx 0.03846$$

Making posterior 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,

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

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.

Problem 2 (10 points)

Now let's manually compute some predictions.

- Given the posterior distribution computed in Problem 1, what is the probability that `40` is also a member of the concept?
- Given the same posterior, what is the probability that `4` is also a member of the concept?

Please **show your work** in the cell below. Your answers from Problem 1 will help you.

Please double check that your equations display properly when printing your homework. We cannot give you credit if your equations are missing. It's your responsibility to check.

YOUR ANSWER GOES HERE

Given,

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

Then using posterior distribution from Problem 1,

$$\begin{aligned} P(40 \in C \mid X) &= \sum_{h \in H} P(40 \in C \mid h) P(h|X) \\ &= P(40 \in C \mid h_1) P(h_1|X) + P(40 \in C \mid h_2) P(h_2|X) \\ &\approx (1)(0.9615) + (1)(0.03846) \approx 1, \\ P(4 \in C \mid X) &= \sum_{h \in H} P(40 \in C \mid h) P(h|X) \\ &= P(4 \in C \mid h_1) P(h_1|X) + P(4 \in C \mid h_2) P(h_2|X) \\ &\approx (0)(0.9615) + (1)(0.03846) \approx 0.03846, \end{aligned}$$

Implementation - Hypothesis space and prior

Let's dive into the implementation of the number game. First, let's import some packages and helpful functions.

`x_all` is the list of all possible numbers. We include 0 as a possible number for programming convenience, although none of the hypotheses include it.

In [1]:
```
from __future__ import print_function
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import random
import numpy as np
from scipy.special import logsumexp

x_max = 100 # maximum number allowed
x_all = np.arange(0,x_max+1)
```

The hypothesis space H includes two main kinds of hypotheses. 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. The second kind consists of interval hypotheses, which are solid intervals of numbers, such as $12, 13, 14, 15, 16, 17$. Each hypothesis will be represented as a list of the numbers that fit that hypothesis. The 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.

The code below shows how to generate the mathematical hypotheses and their prior probabilities (in natural log space). To keep the prior simple, each mathematical hypothesis is given equal weight in the prior.

In [2]:

```
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, 67, 71, 73, 79, 83, 89, 97]

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_primes()]
    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, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73,
75, 77, 79, 81, 83, 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, 3
8, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72,
74, 76, 78, 80, 82, 84, 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]
```

All possible interval hypotheses and their prior probabilities can be generated with the following code. All interval hypotheses are not equally likely in the prior, and following Tenenbaum's specification, we use an Erlang distribution with parameter `sigma=10` to express an expectation for intermediate-sized hypotheses.

In [3]:
```
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]
```

Together, we can define a `generate_hypotheses` function that uses all of this code to generate the complete set of hypotheses and their prior probabilities. We also use `convert_h_list_to_numpy` to convert each hypothesis from a Python list of numbers to a binary numpy array for speedier Bayesian computations later (run code to see example).

In [4]:
```
def convert_h_list_to_numpy(h_list):
    h_numpy = np.zeros(x_all.size)
    h_numpy[np.array(h_list)] = 1
    return h_numpy

def generate_hypotheses(mylambda):
    h_math,lp_math = generate_math_hypotheses(mylambda)
    h_interval,lp_interval = generate_interval_hypotheses(mylambda)
    H = h_math + h_interval
    H_numpy = [convert_h_list_to_numpy(h) for h in H]
    log_prior = np.concatenate((lp_math,lp_interval))
    assert(np.isclose(np.sum(np.exp(log_prior)),1.0))
    return H_numpy,log_prior

print("Example of converting list hypothesis to numpy array...")
print("original hypothesis:")
h_list = [2,4,6]
print(h_list)
h_numpy = convert_h_list_to_numpy(h_list)
print("converted numpy array:")
print(h_numpy[0:10])
```

```
Example of converting list hypothesis to numpy array...
original hypothesis:
[2, 4, 6]
converted numpy array:
[0. 0. 1. 0. 1. 0. 1. 0. 0. 0.]
```

Implementation - Posterior and likelihood

Now we need code to do the Bayesian computations, including a function `log_posterior` and `log-likelihood` . For probabilistic modeling, we like to compute probabilities in log-space to help avoid numerical issues such as underflow. Study the function `log_posterior` to make sure you understand how it works. Also, see the nifty `logsumexp` function (see scipy doc (https://docs.scipy.org/doc/scipy-0.19.0/reference/generated/scipy.misc.logsumexp.html)) which is used to normalize log-probability distributions in a numerically safer way.

In [5]:
```
def log_posterior(data,list_hypothesis,log_prior):
    # INPUT
    #  data : python list of observed numbers (X)
    #  list_hypothesis : [nh length list] each hypothesis is a binary nu
mpy array
    #  log_prior : numpy vector [length nh] (log prior value for each hy
pothesis)
    data_numpy = convert_h_list_to_numpy(data) # length nh numpy vector
    nh = len(list_hypothesis)
    ll = np.zeros(nh)
    for idx,h in enumerate(list_hypothesis):
        ll[idx] = log_likelihood(data_numpy,h)
    lpost = ll + log_prior
    lpost = lpost - logsumexp(lpost)
    return lpost
```

Problem 3 (10 points)

Fill in the missing code below to complete the `log-likelihood` function.

In [38]:
```
def log_likelihood(data_numpy, hypothesis):
    # INPUT
    #  data_numpy : size x_max binary numpy array (observed numbers)
    #  hypothesis: size x_max binary numpy array (included numbers in si
ngle hypothesis)
    # RETURN
    #  ll : log-likelihood value (REMEMBER TO CONVERT TO NATURAL LOG (n
p.log()))
    assert(hypothesis.size == data_numpy.size)
    n_d = np.sum(data_numpy)
    n_h = np.sum(hypothesis)

    # TODO: Add your code to check whether or not the hypothesis contain
s all of the data.
    #   If it does not, return -np.inf (log(0))
    #   raise Exception('Replace with your code.')
    for i in range(len(data_numpy)):
        if data_numpy[i] != hypothesis[i]:
            if data_numpy[i] == 1:
                return -np.inf

    # TODO: Add your code to compute the log-likelihood if the hypothesi
s contains all of the data.
    # raise Exception('Replace with your code.')
    ll = np.log( (1/n_h)**n_d )

    return ll
```

Implementation - Making Bayesian predictions

We now have all the code in place to make Bayesian predictions regarding the membership of new numbers, as described by the previous equations,

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

Problem 4 (10 points)

- Fill in the missing code below to help complete the `bayesian_predictions` function.
- Use the `bayesian_predictions` function to double check your answer in Problem 2. Remember that there are only two hypotheses "multiples of 10" and "even numbers". For X, the numbers 10 and 30 were observed. Compute the probability that 40 and 4 are a member of the same concept as the numbers in X. Don't forget to convert your individual hypotheses to numpy arrays using `convert_h_list_to_numpy`

The answer in Problem 2 is correct. I am getting the same posterior probabilities.

In [26]:
```
def bayesian_predictions(data_eval, data, list_hypothesis, log_prior):
    # INPUT
    #  data_eval : [length ne python list] of new numbers we want to che
ck the probability of membership for
    #     each number in data_eval is to be evaluated independently -- i
t's a separate 'y' in equation above
    #  data : [python list] observed numbers (X)
    #  list_hypothesis : python list of hypotheses, each is a binary num
py array
    #  log_prior : numpy vector [length nh] which is the log prior value
for each hypothesis
    #
    # RETURN
    #  pp : numpy vector [size ne] of predicted probabilities of new num
bers in data_eval (NOTE: NOT IN LOG SPACE)
    lpost = log_posterior(data,list_hypothesis,log_prior)
    post = np.exp(lpost) # posterior probabilities
    h_mat = np.array(list_hypothesis) # create a [nh by x_max] numpy mat
rix, showing numbers in each hypothesis
    ne = len(data_eval) # how many numbers to evaluate
    pp = np.zeros(ne) # predicted probability of each number
    for idx,de in enumerate(data_eval):
        #TODO : Add your code here to compute predicted probabilities. C
an be a single line with form "pp[idx] = "..
        #raise Exception('Replace with your code.')
        predicted_prob = 0
        for i, hypothesis in enumerate(h_mat):
            if h_mat[i][de] == 1:
                predicted_prob += post[i]
        pp[idx] = predicted_prob

    return pp

# TODO : Check your answers for problem 2 using the bayesian_prediction
 function
data_eval = [40, 4]
data = [10, 30]
h1 = make_h_mult_of_y(10)
h2 = make_h_even()
list_hypothesis = [convert_h_list_to_numpy(h1), convert_h_list_to_numpy(
h2)]
log_prior = np.log([1/2, 1/2])
bayesian_predictions(data_eval, data, list_hypothesis, log_prior)
```

Out[26]:
```
array([1.        , 0.03846154])
```

Running the model and predicting human data

Now we have all the pieces in place to run the complete number game model.

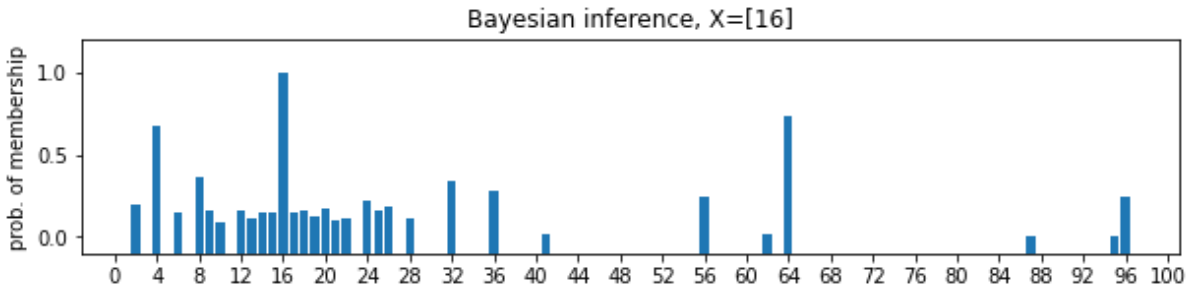
First, let's described the data from participants. Tenenbaum ran eight participants in an experiment where they were provided with various sets X of random positive examples from a concept. They were asked to rate the probability that each of 30 test numbers would belong to the same concept of the observed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

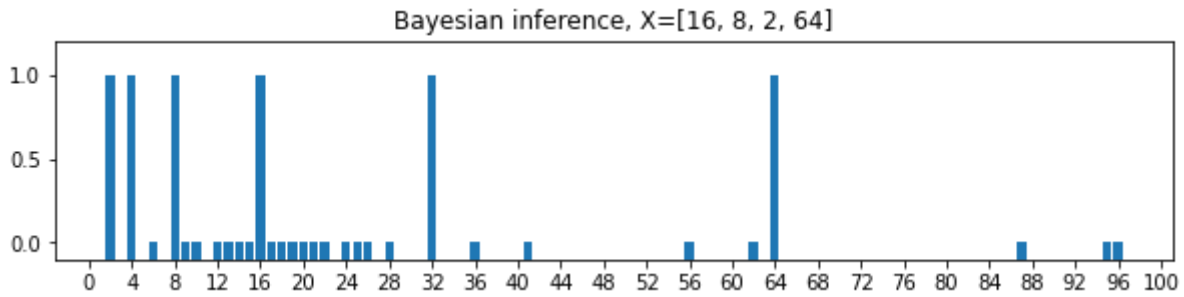
The following plot shows the mean rating across the human participants for three different sets. Note that since only 30 test numbers were evaluated, and thus a value of 0 in the plot indicates missing data (rather than zero probability).

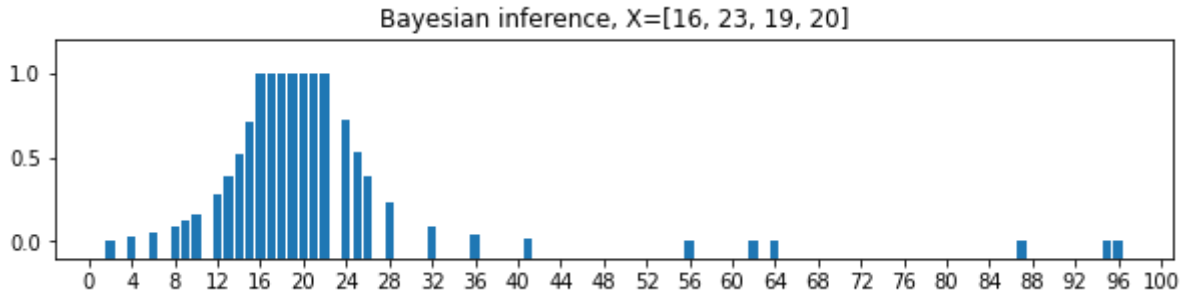
Let's produce the same plots for the Bayesian concept learning model using the code below.

In [39]:
```
def plot_predictions(x_eval,mypred):
    mybottom = -0.1
    plt.figure(figsize=(10, 2))
    plt.bar(x_eval,mypred-mybottom,bottom=mybottom)
    plt.ylim((mybottom,1.2))
    plt.xticks(np.arange(0, x_max+1, step=4))
    plt.yticks([0,0.5,1])
```

In [40]:
```
H_all,log_prior_all = generate_hypotheses(mylambda=2./3)
x_eval = [2,4,6,8,9,10]+list(range(12,23))+[24,25,26,28,32,36,41,56,62,6
4,87,95,96]

mypred = bayesian_predictions(x_eval, [16], H_all, log_prior_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16]')
plt.ylabel('prob. of membership')
mypred = bayesian_predictions(x_eval, [16, 8, 2, 64], H_all, log_prior_a
ll)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 8, 2, 64]')
mypred = bayesian_predictions(x_eval, [16, 23, 19, 20], H_all, log_prior
_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 23, 19, 20]')
plt.show()
```







Problem 5 (10 points)

Using the code above, produce plots the show the Bayesian model's predictions. Produce the plots for the default `mylambda = 2./3`, but also reproduce the plots below for another value of `mylambda` that shows qualitatively different results.

- Which value of `mylambda` seems to capture the human behavior data the best?
- Comment on why the predictions change for different values of `mylambda`.

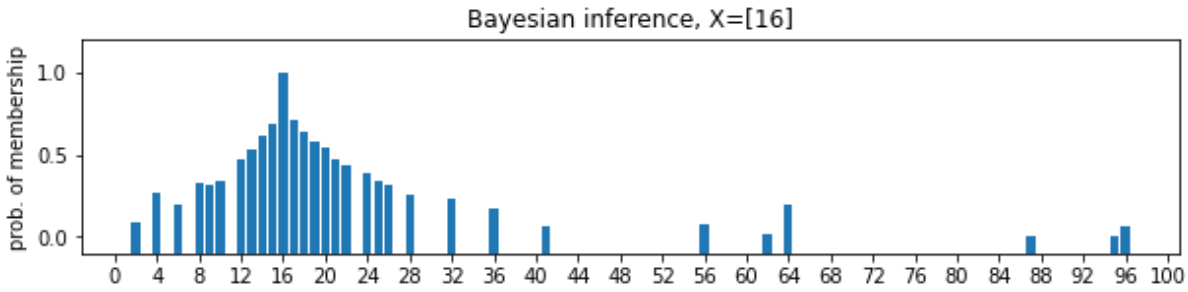
Your response in the cells below should include plots for a least one new setting of lambda, and 1-2 paragraphs of discussion about how the results change as a function of `mylambda`.

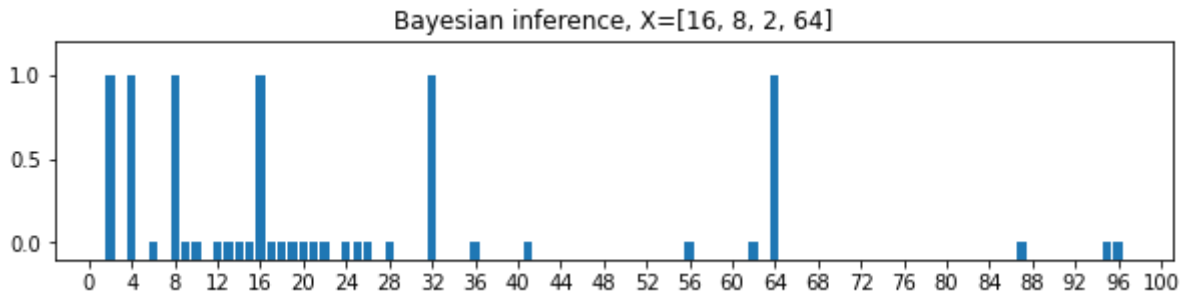
YOUR RESPONSE GOES HERE

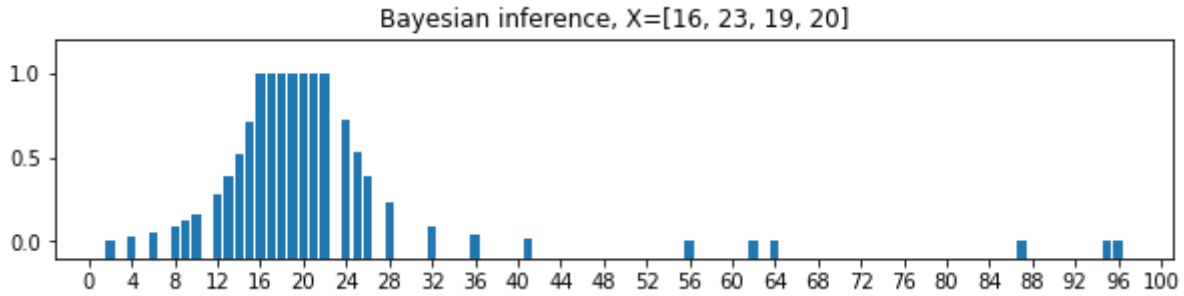
The best mylambda value was 0.5, which the Bayesian model most precisely captured the human behavior. The graphs below with mylambda = {.1, .5, .9} gives an additional idea that higher lambda (i.e. .9) performs better than lower labmda value (i.e. .1). This means human cognition prefers the equal mix of mathematical and interval hypothesis. And between the mathematical and interval hypothesis, human behavior tends to follow mathematical hypothesis more.

The reason why the predictions change for different values of mylambda is coming from the fact that mylambda value controls how much of the prior is specified by each type of hypothesis. Since the weight of mathematical and interval hypotheses change, the overall Bayesian inference probabilities change as well.

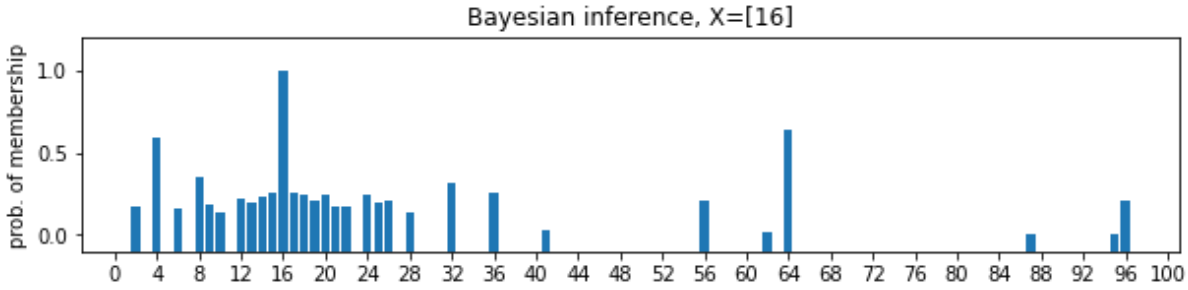
In [41]:
```
H_all,log_prior_all = generate_hypotheses(mylambda=0.1)
x_eval = [2,4,6,8,9,10]+list(range(12,23))+[24,25,26,28,32,36,41,56,62,6
4,87,95,96]

mypred = bayesian_predictions(x_eval, [16], H_all, log_prior_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16]')
plt.ylabel('prob. of membership')
mypred = bayesian_predictions(x_eval, [16, 8, 2, 64], H_all, log_prior_a
ll)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 8, 2, 64]')
mypred = bayesian_predictions(x_eval, [16, 23, 19, 20], H_all, log_prior
_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 23, 19, 20]')
plt.show()
```

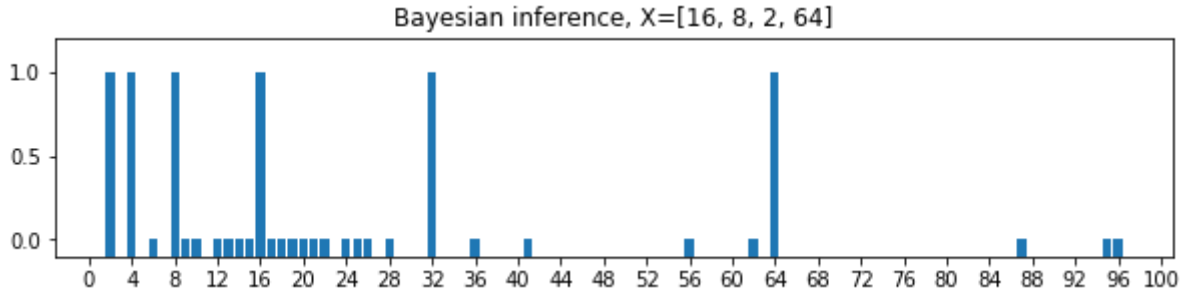


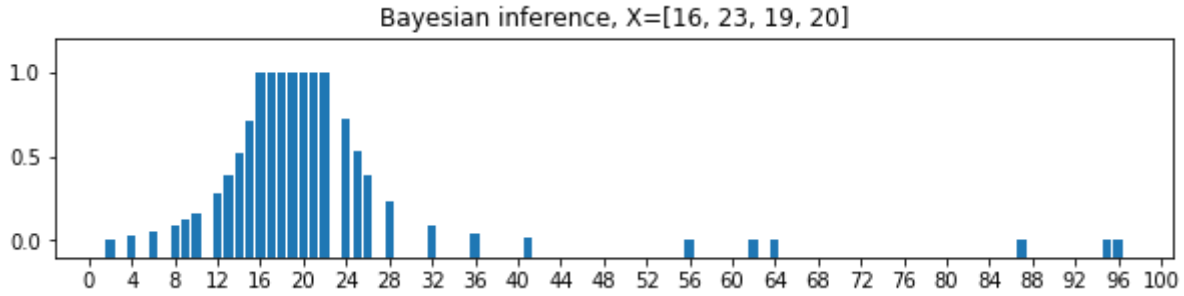




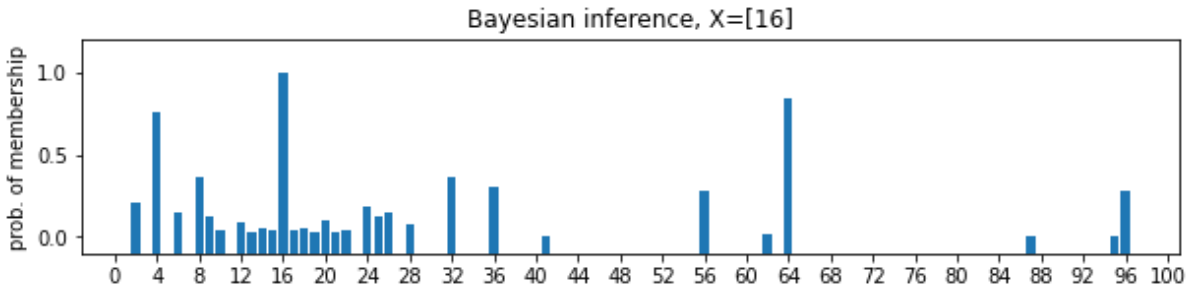
In [44]:
```
H_all,log_prior_all = generate_hypotheses(mylambda=0.5)
x_eval = [2,4,6,8,9,10]+list(range(12,23))+[24,25,26,28,32,36,41,56,62,6
4,87,95,96]

mypred = bayesian_predictions(x_eval, [16], H_all, log_prior_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16]')
plt.ylabel('prob. of membership')
mypred = bayesian_predictions(x_eval, [16, 8, 2, 64], H_all, log_prior_a
ll)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 8, 2, 64]')
mypred = bayesian_predictions(x_eval, [16, 23, 19, 20], H_all, log_prior
_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 23, 19, 20]')
plt.show()
```

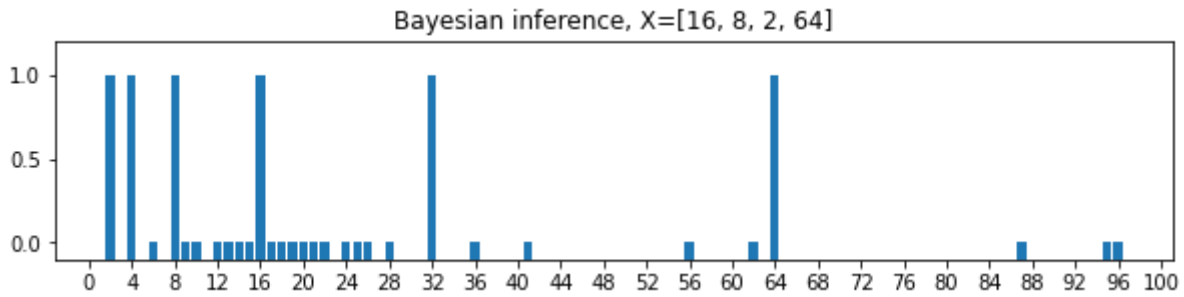


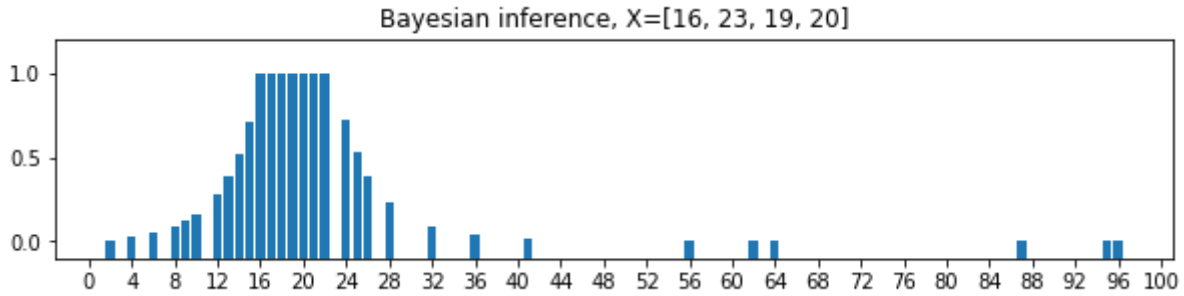




In [42]:
```
H_all,log_prior_all = generate_hypotheses(mylambda=0.9)
x_eval = [2,4,6,8,9,10]+list(range(12,23))+[24,25,26,28,32,36,41,56,62,6
4,87,95,96]

mypred = bayesian_predictions(x_eval, [16], H_all, log_prior_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16]')
plt.ylabel('prob. of membership')
mypred = bayesian_predictions(x_eval, [16, 8, 2, 64], H_all, log_prior_a
ll)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 8, 2, 64]')
mypred = bayesian_predictions(x_eval, [16, 23, 19, 20], H_all, log_prior
_all)
plot_predictions(x_eval,mypred)
plt.title('Bayesian inference, X=[16, 23, 19, 20]')
plt.show()
```







Making maximum a posteriori (MAP) predictions

As implemented above, to make proper Bayesian predictions, we marginalize (average) over all of our hypotheses weighted by our posterior belief.

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

However, the summation over all hypotheses isn't always tractable (although it is in this case). A common approximation to full Bayesian inference is called maximum a posteriori (MAP) inference: making predictions based on just the best hypothesis h^*, as determined by its score under the posterior distribution.

$$h^* = \text{argmax}_{h \in H} P(h^*|X).$$

Then, predictions are made as follows:

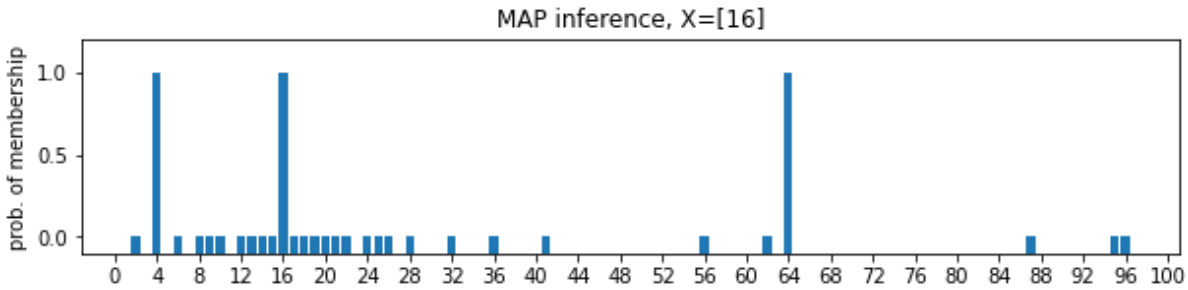
$$P(y \in C \mid X) \approx P(y \in C \mid h^*).$$

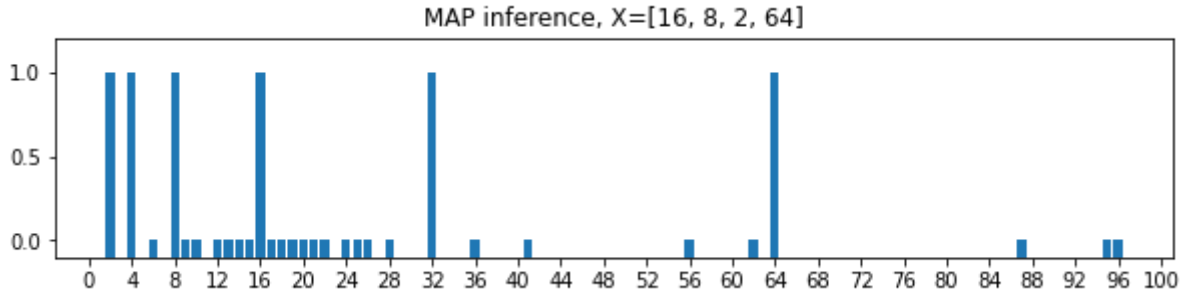
In essence, we pretend that there is only one term in our hypotheses average. This can be a good approximation if just one hypothesis is dominant in the posterior. Otherwise, it can be a poor approximation.

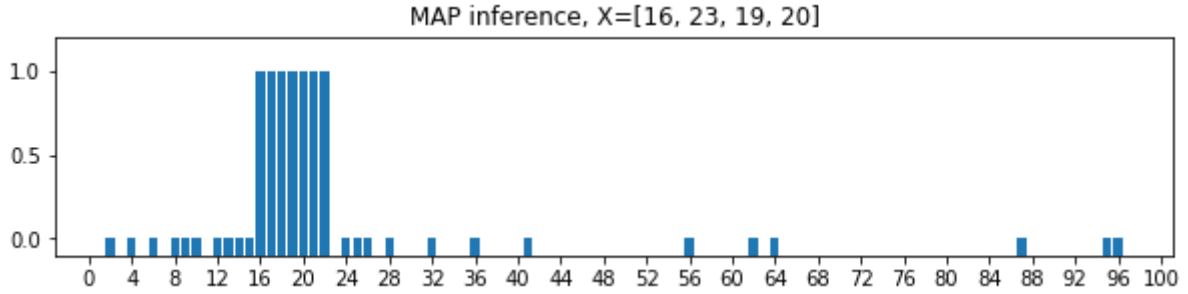
Problem 6 (10 points)

- Fill in the missing code below to help complete the `MAP_predictions` function.

In [47]:
```
def MAP_predictions(data_eval, data, list_hypothesis, log_prior):
    # INPUT
    #  data_eval : [length ne python list] of new numbers we want to che
ck the probability of membership for
    #     each number in data_eval is to be evaluated independently -- i
t's a separate 'y' in equation above
    #  data : [python list] observed numbers (X)
    #  list_hypothesis : python list of hypotheses, each is a binary num
py array
    #  log_prior : numpy vector [length nh] which is the log prior value
for each hypothesis
    #
    # RETURN
    #  pp : numpy vector [size ne] of predicted probabilities of new num
bers in data_eval (NOTE: NOT IN LOG SPACE)
    lpost = log_posterior(data,list_hypothesis,log_prior)
    ne = len(data_eval) # how many numbers to evaluate
    pp = np.zeros(ne) # predicted probability of each number
    #TODO : Add your code here to compute MAP approximation
    # raise Exception('Replace with your code.')

    lpost = log_posterior(data,list_hypothesis,log_prior)
    post = np.exp(lpost) # posterior probabilities
    hyp_max = list_hypothesis[np.argmax(post)]
    #h_mat = np.array(list_hypothesis) # create a [nh by x_max] numpy ma
trix, showing numbers in each hypothesis
    ne = len(data_eval) # how many numbers to evaluate
    pp = np.zeros(ne) # predicted probability of each number


    for idx,de in enumerate(data_eval):
        #TODO : Add your code here to compute predicted probabilities. C
an be a single line with form "pp[idx] = "..
        #raise Exception('Replace with your code.')
        if hyp_max[de] == 1:
            pp[idx] = 1
    return pp

H_all,log_prior_all = generate_hypotheses(mylambda=2./3)
x_eval = [2,4,6,8,9,10]+list(range(12,23))+[24,25,26,28,32,36,41,56,62,6
4,87,95,96]

mypred = MAP_predictions(x_eval, [16], H_all, log_prior_all)
plot_predictions(x_eval,mypred)
plt.title('MAP inference, X=[16]')
plt.ylabel('prob. of membership')
mypred = MAP_predictions(x_eval, [16, 8, 2, 64], H_all, log_prior_all)
plot_predictions(x_eval,mypred)
plt.title('MAP inference, X=[16, 8, 2, 64]')
mypred = MAP_predictions(x_eval, [16, 23, 19, 20], H_all, log_prior_all)
plot_predictions(x_eval,mypred)
plt.title('MAP inference, X=[16, 23, 19, 20]')
plt.show()
```







Problem 7 (5 points)

Using the code above, produce plots that show the model's predictions using the MAP approximation. Is MAP inference or full Bayesain inference a better account of the human behavioral data in this experiment? Why? Your answer should just be a few sentences.

YOUR RESPONSE GOES HERE

Full Bayesian inference is a better account of the human behavioral data because human cognition often times doesn't just rely on a single hypothesis. As a result, the combination of math and interval hypotheses gave the closest prediction to human behavioral data. When we only account for the hypothesis with maximum probability, it is possible to miss an important part of the data. Since full Bayesian inference takes into account more data and information, it is closer to how human cognition behavior.

Problem 8 (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

I believe this model is convincing in its ability to understand human judgment because the inference graphs are able to closely resemble the human prediction probability graphs. This result makes sense because the numbers game is similar to how humans learn from the environments in childhood and even in the adult stages of life. With the vast amount of varying factors that govern the society, it is impossible for humans to have all the necessary information; instead, we are forced to make predictions based on our previous experiences and update our decision making. This process is very similar to Bayesian modeling involving prior, likelihood, and posterior, where we use prior knowledge to improve decision making with likelihood and finally find the posterior probability. Since process is very similar to human decision making, Bayesian is able to capture the human behavior in a convincing level.

Important part of Bayesian process is hypothesis space. In terms of human behavior, hypothesis space is developed from the experiences and memories that can possibly match with the phenomenon observed (i.e. output numbers in the number game). Since machines cannot clearly know which hypothesis space to use and incorporate, we manually input the hypotheses in the Bayesian model. However, using algorithms like KNN and clustering, we may be able to approximate Bayesian inference rather than enumerating all the hypotheses.

Problem 9 (20 points)

Here we consider a sampling-based strategy to approximate full Bayesian inference (as opposed to the cruder MAP approximation). This problem asks you to implement a "likelihood weighted sampler" as discussed in lecture. Review the lecture notes on importance sampling and specifically "likelihood weighted sampling", where we choose the approximate distribution Q to be the prior distribution over hypotheses.

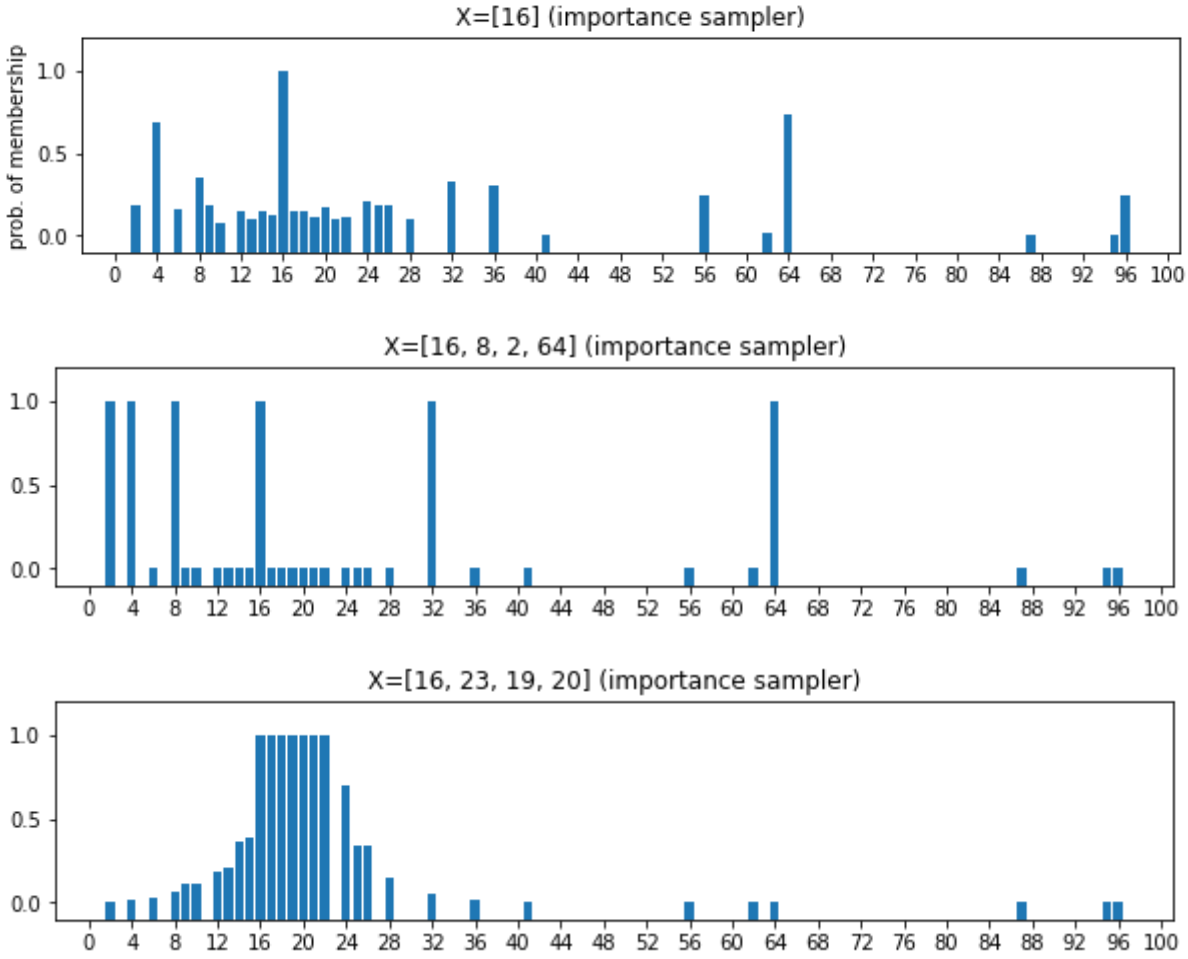
- Fill in the missing code below to help complete the `draw_prior_samples`, `weight_samples`, and `importance_sampler_predictions` functions.
- Run your likelihood weighted sampler for 2000 samples and reproduce the plots in Problem 5. Does approximate inference match the exact inference?

Approximate inference does match the exact inference.

In [49]:
```
def draw_prior_samples(nsamp):
    # INPUT
    #  nsamp : number of hypotheses to be sampled from the prior
    #
    # RETURN
    #  list_H : [nsamp length python list] of sampled hypotheses (each i
s a binary numpy vector [length x_max])
    #
    # TODO: Add your code to draw a list of samples (from H_all) given t
heir prior probabilities (log_prior_all)
    # raise Exception('Replace with your code.')
    H_all, log_prior_all = generate_hypotheses(mylambda=2./3)
    sample=np.random.choice(len(H_all), size=nsamp, replace=True, p=np.e
xp(log_prior_all))
    list_H = []
    for i in sample:
        list_H.append(H_all[i])
    return list_H

def weight_samples(data, list_H):
    # INPUT
    #  data : [python list] of observed data
    #  list_H : [nsamp length pythong list] of sampled hypotheses (each
 is a binary numpy vector [length x_max])
    #     these are the sampled "particles" in the importance sampler
    #
    # Output
    #  log_wt : numpy vector [size nsamp] log importance weight of each
 sample in list_H
    #
    # TODO: Add your code to return the log-weight of each particle
    # raise Exception('Replace with your code.')
    log_wt = []
    for hyp in list_H:
        log_wt.append(log_likelihood(convert_h_list_to_numpy(data),hyp))
    return log_wt

def importance_sampler_predictions(data_eval, list_H, log_wt):
    # INPUT
    #  data_eval : [length ne python list] of new numbers we want to eva
luate the probability of membership for
    #     each number in data_eval is to be evaluated independently (as
 in bayesian_predictions function)
    #  list_H : nsamp length pythong list] of sampled hypotheses (each i
s a binary numpy vector [length x_max])
    #  log_wt : numpy vector [size nsamp] log importance weight of each
 sample in list_H
    #
    # RETURN
    #  pp : numpy vector [size ne] of predicted probabilities of new num
bers  (NOTE: NOT IN LOG SPACE)
    wt = np.exp(log_wt)
    h_mat = np.array(list_H) # create a [nsamp by x_max] numpy matrix, s
howing numbers in each hypothesis/sample
    ne = len(data_eval) # how many numbers to evaluate
    pp = np.zeros(ne) # predicted probability of each number
```

```
    for idx,de in enumerate(data_eval):
        #TODO : Add your code here/ Can be a single line with form "pp[i
dx] = "..
        # raise Exception('Replace with your code.')
        predicted_prob = 0
        for i, hypothesis in enumerate(h_mat):
            if h_mat[i][de] == 1:
                predicted_prob += wt[i]
        pp[idx] = predicted_prob / np.sum(wt)

    return pp

nsamples_importance = 2000 # number of samples
plt.figure()
list_H_importance = draw_prior_samples(nsamples_importance) # prior samp
les can be re-used across queries
log_wt_importance = weight_samples([16],list_H_importance)
mypred = importance_sampler_predictions(x_eval, list_H_importance, log_w
t_importance)
plot_predictions(x_eval,mypred)
plt.title('X=[16] (importance sampler)')
plt.ylabel('prob. of membership')

log_wt_importance = weight_samples([16, 8, 2, 64],list_H_importance)
mypred = importance_sampler_predictions(x_eval, list_H_importance, log_w
t_importance)
plot_predictions(x_eval,mypred)
plt.title('X=[16, 8, 2, 64] (importance sampler)')

log_wt_importance = weight_samples([16, 23, 19, 20],list_H_importance)
mypred = importance_sampler_predictions(x_eval, list_H_importance, log_w
t_importance)
plot_predictions(x_eval,mypred)
plt.title('X=[16, 23, 19, 20] (importance sampler)')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



In [ ]:

Homework - Bayesian modeling - Part B (40 points)

Probabilistic programs for productive reasoning

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

This homework is due before midnight on Monday, April 4.

People can reason in very flexible and sophisticated ways. Let's consider an example that was introduced in Gerstenberg and Goodman (2012; see below for reference). Imagine that Brenden and Todd are playing tennis together, and Brenden wins the game. You might suspect that Brenden is a strong player, but you may also not think much of it, since it was only one game and we don't know much about Todd's ability.

Now imagine that you also learn that Todd has recently played against two other faculty members in the Psychology department, and he won both of those games. You would now have a higher opinion of Brenden's skill.

Now, say you also learn that Todd was feeling very lazy in his game against Brenden. This could change your opinion yet again about Brenden's skill.

In this notebook, you will get hands on experience using simple probabilistic programs and Bayesian inference to model these patterns of reasoning. Probabilistic programs are a powerful way to write Bayesian models, and they are especially useful when the prior distribution is more complex than a list of hypotheses, or is inconvenient to represent with a probabilistic graphical model.

Probabilistic programming is an active area of research. There are many specially designed probabilistic programming languages such as WebPPL (http://webppl.org/) and Church (http://v1.probmods.org/). Recently, new languages have been introduced that combine aspects of probabilistic programming and neural networks, such as Pyro (http://pyro.ai/), and Edward (http://edwardlib.org/). Rather than using a particular language, we will use vanilla Python to express an interesting probability distribution as a probabilistic program, and you will be asked to write your own rejection sampler for inference. More generally, an important component of the appeal of probabilistic programming is that when using a specialized language, you can take advantage of general algorithms for Bayesian inference without having to implement your own.

Great, let's proceed with the probabilistic model of tennis!

The Bayesian tennis game was introduced by Tobi Gerstenberg and Noah Goodman in the following material:

- Gerstenberg, T., & Goodman, N. (2012). Ping Pong in Church: Productive use of concepts in human probabilistic inference. In Proceedings of the Annual Meeting of the Cognitive Science Society.
- Probabilistic models of cognition online book (Chapter 3) (https://probmods.org/chapters/03-conditioning.html)

Probabilistic model

The generative model can be described as follows. There are various players engaged in a tennis tournament. Matches can be played either as a singles match (Player A vs. Player B) or as a doubles match (Player A and Player B vs. Player C and Player D).

Each player has a latent `strength` value which describes his or her skill at tennis. This quantity is unobserved for each player, and it is a persistent property in the world. Therefore, the `strength` stays the same across the entire set of matches.

A match is decided by whichever team has more `team_strength`. Thus, if it's just Player A vs. Player B, the stronger player will win. If it's a doubles match, `team_strength` is the sum of the strengths determines which team will be the `winner`. However, there is an additional complication. On occasion (with probability 0.1), a player becomes `lazy`, in that he or she doesn't try very hard for this particular match. For the purpose of this match, his or her `strength` is reduced by half. Importantly, this is a temporary (non-persistent) state which is does not effect the next match.

This completes our generative model of how the data is produced. In this assignment, we will use Bayesian inference to reason about latent parameters in the model, such as reasoning about a player's strength given observations of his or her performance.

Concepts as programs

A powerful idea is that we can model concepts like `strength`, `lazy`, `team_strength`, `winner`, and `beat` as programs, usually simple stochastic functions that operate on inputs and produce outputs. You will see many examples of this in the code below. Under this view, the meaning of a "word" comes from the semantics of the program, and how the program interact with eachother. Can all of our everyday concepts be represented as programs? It's an open question, and the excitement around probabilistic programming is that it provides a toolkit for exploring this idea.

In [1]:
```
# Import the necessary packages
from __future__ import print_function
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import random
import numpy as np
from scipy.stats.mstats import pearsonr
```

Persistent properties

The strength of each player is the only persistent property. In the code below, we create a `world` class which stores the persistent states. In this case, it's simply a dictionary `dict_strength` that maps each player's name to his or her strength. Conveniently, the world class gives us a method `clear` that resets the world state, which is useful when we want to clear everything and produce a fresh sample of the world.

The `strength` function takes a player's `name` and queries the world `W` for the appropriate strength value. If it's a new player, their strength is sampled from a Gaussian distribution (with $\mu = 10$ and $\sigma = 3$) and stored persistently in the world state. As you can see, this captures something about our intuitive notion of strength as a persistent property.

In [2]:
```
class world():
    def __init__(self):
        self.dict_strength = {}
    def clear(self): # used when sampling over possible world
        self.dict_strength = {}

W = world()

def strength(name):
    if name not in W.dict_strength:
        W.dict_strength[name] = abs(random.gauss(10,3))
    return W.dict_strength[name]
```

Computing team strength

Next is the `lazy` function. When the lazy function is called on the `name` of a particular player, the answer is computed fresh each time (and is not stored persistently like strength).

The total strength of a team `team_strength` takes a list of names `team` and computes the aggregate strength. This is a simple sum across the team members, with a special case for lazy team members. For a game like tennis, this program captures aspects of what we mean when we think about "the strength of a team" -- although simplified, of course.

In [3]:
```
def lazy(name):
    return random.random() < 0.1
```

In [4]:
```
def team_strength(team):
    # team : list of names
    mysum = 0.
    for name in team:
        if lazy(name):
            mysum += (strength(name) / 2.)
        else:
            mysum += strength(name)
    return mysum
```

Computing the winner

The `winner` of a match returns the team with a higher strength value. Again, we can represent this as a very simple function of `team_strength`.

Finally, the function `beat` checks whether `team1` outperformed `team2` (returning `True`) or not (returning `False`).

In [5]:
```
def winner(team1,team2):
    # team1 : list of names
    # team2 : list of names
    if team_strength(team1) > team_strength(team2):
        return team1
    else:
        return team2

def beat(team1,team2):
    return winner(team1,team2) == team1
```

Probabilistic inference

Problem 1 (15 points)

Your first task is to complete the missing code in the `rejection_sampler` function below to perform probabilistic inference in the model. You give it a list of function handles `list_f_conditions` which represent the data we are conditioning on, and thus these functions must evaluate to `True` in the current state of the world. If they do, then you want to grab the variable of interest using the function handle `f_return` and store it in the `samples` vector, which is returned as a numpy array. Please fill out the function below.

Note: A function handle `f_return` is a pointer to a function which can be executed with the syntax `f_return()`. We need to pass handles, rather than pre-executed functions, so the rejection sampler can control for itself when to execute the functions.

In [6]:
```
def rejection_sampler(f_return, list_f_conditions, nsamp=10000):
    # Input
    #  f_return : function handle that grabs the variable of interest wh
en executed
    #  list_f_conditions: list of conditions (function handles) that we
 are assuming are True
    #  nsamp : number of attempted samples (default is 10000)
    # Output
    #  samples : (as a numpy-array) where length is the number of actua
l, accepted samples
    samples = []
    for i in range(nsamp):
        # TODO : your code goes here (don't forget to call W.clear() bef
ore each attempted sample)
        W.clear()
        check_true = True
        for func in list_f_conditions:
            if func() == False:
                check_true = False
        if check_true == True:
            samples.append(f_return())
    return np.array(samples)
```

Use the code below to test your rejection sampler. Let's assume Bob and Mary beat Tom and Sue in their tennis match. Also, Bob and Sue beat Tom and Jim. What is our mean estimate of Bob's strength? (The right answer is around 11.86, but you won't get that exactly. Check that you are in the same ballpark).

In [7]:
```
f_return = lambda : strength('bob')
list_f_conditions = [lambda : beat( ['bob', 'mary'],['tom', 'sue'] ), la
mbda : beat( ['bob', 'sue'],  ['tom', 'jim'] )]
samples = rejection_sampler(f_return, list_f_conditions, nsamp=50000)
mean_strength = np.mean(samples)
print("Estimate of Bob's strength: mean = " + str(mean_strength) + "; ef
fective n = " + str(len(samples)))
```

```
Estimate of Bob's strength: mean = 11.861837497815158; effective n =
177
```

Comparing judgments from people and the model

We want to explore how well the model matches human judgments of strength. In the table below, there are six different doubles tennis tournaments. Each tournament consists of three doubles matches, and each letter represents a different player. Thus, in the first tournament, the first match shows Player A and Player B winning against Player C and Player D. In the second match, Player A and Player B win against Player E and F. Given the evidence, how strong is Player A in Scenario 1? How strong is Player A in Scenario 2? The data in the different scenarios should be considered separate (they are alternative possible worlds, rather than sequential tournaments).

> For each tournament, rate how strong you think Player A is using a 1 to 7 scale, where 1 is the weakest and 7 is the strongest. Also, explain the scenario to a friend and ask for their ratings as well. Be sure to mention that sometimes a player is lazy (about 10 percent of the time) and doesn't perform as well.

In [8]:
```
# TODO : YOUR DATA GOES HERE
subject1_pred = np.array([5,6,6,4,6,7])
subject2_pred = np.array([4,5,6,6,7,7])
```

The code below will use your rejection sampler to predict the strength of Player A in all six of the scenarios. These six numbers will be stored in the array `model_pred`

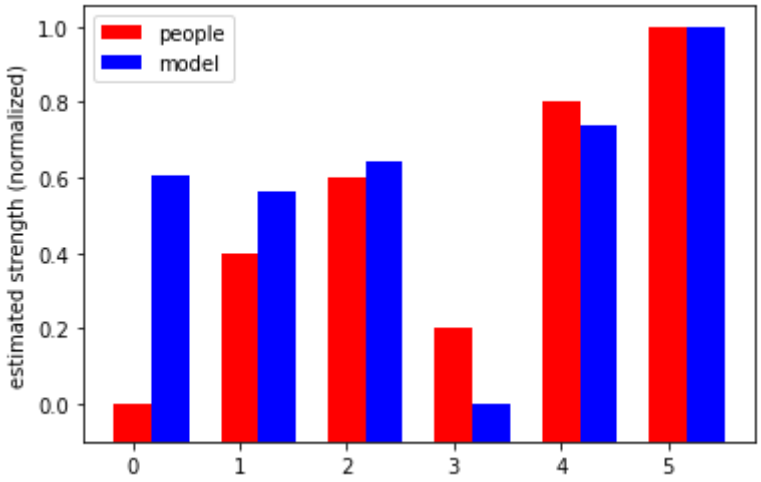
In [9]:

```
model_pred = []

f_return = lambda : strength('A')

f_conditions = [lambda : beat( ['A', 'B'],['C', 'D'] ), lambda : beat( [
'A', 'B'],['E', 'F'] ), lambda : beat( ['A', 'B'],  ['G', 'H'] ) ]
samples = rejection_sampler(f_return, f_conditions)
print("Scenario 1")
print("  sample mean : " + str(np.mean(samples)) + "; n=" + str(len(samp
les)))
model_pred.append(np.mean(samples))

f_conditions = [lambda : beat( ['A', 'B'],['E', 'F'] ), lambda : beat( [
'A', 'C'],['E', 'G'] ), lambda : beat( ['A', 'D'],  ['E', 'H'] ) ]
samples = rejection_sampler(f_return, f_conditions)
print("Scenario 2")
print("  sample mean : " + str(np.mean(samples)) + "; n=" + str(len(samp
les)))
model_pred.append(np.mean(samples))

f_conditions = [lambda : beat( ['A', 'B'],['E', 'F'] ), lambda : beat([
'E', 'F'], ['B', 'C'] ), lambda : beat( ['E', 'F'], ['B', 'D'] ) ]
samples = rejection_sampler(f_return, f_conditions)
print("Scenario 3")
print("  sample mean : " + str(np.mean(samples)) + "; n=" + str(len(samp
les)))
model_pred.append(np.mean(samples))

f_conditions = [lambda : beat( ['A', 'B'],['E', 'F'] ), lambda : beat( [
'B', 'C'],['E', 'F'] ), lambda : beat( ['B', 'D'],  ['E', 'F'] ) ]
samples = rejection_sampler(f_return, f_conditions)
print("Scenario 4")
print("  sample mean : " + str(np.mean(samples)) + "; n=" + str(len(samp
les)))
model_pred.append(np.mean(samples))

f_conditions = [lambda : beat( ['A', 'B'],['E', 'F'] ), lambda : beat( [
'A', 'C'],['G', 'H'] ), lambda : beat( ['A', 'D'],  ['I', 'J'] ) ]
samples = rejection_sampler(f_return, f_conditions)
print("Scenario 5")
print("  sample mean : " + str(np.mean(samples)) + "; n=" + str(len(samp
les)))
model_pred.append(np.mean(samples))

f_conditions = [lambda : beat( ['A', 'B'],['C', 'D'] ), lambda : beat( [
'A', 'C'],['B', 'D'] ), lambda : beat( ['A', 'D'],  ['B', 'C'] ) ]
samples = rejection_sampler(f_return, f_conditions)
print("Scenario 6")
print("  sample mean : " + str(np.mean(samples)) + "; n=" + str(len(samp
les)))
model_pred.append(np.mean(samples))
```

```
Scenario 1
  sample mean : 12.079430666758954; n=2146
Scenario 2
  sample mean : 11.96976391904173; n=2215
Scenario 3
  sample mean : 12.172242731434608; n=743
Scenario 4
  sample mean : 10.535158536445117; n=2740
Scenario 5
  sample mean : 12.416685235284469; n=1689
Scenario 6
  sample mean : 13.074950254182998; n=1262
```

This code creates a bar graph to compare the human and model predictions for Player A's strength.

In [10]:
```
def normalize(v):
    # scale vector v to have min 0 and max 1
    v = v - np.min(v)
    v = v / np.max(v)
    return v

human_pred_norm = normalize((subject1_pred+subject2_pred)/2.)
model_pred_norm = normalize(model_pred)

# compare predictions from people vs. Bayesian mdoel
mybottom = -0.1
width = 0.35
plt.figure(1)
plt.bar(np.arange(len(human_pred_norm)),human_pred_norm-mybottom, width,
bottom=mybottom, color='red')
plt.bar(np.arange(len(human_pred_norm))+width, model_pred_norm-mybottom,
width, bottom=mybottom, color='blue')
plt.ylabel('estimated strength (normalized)')
plt.legend(('people','model'))
plt.show()

r = pearsonr(human_pred_norm,model_pred_norm)[0]
print('correlation between human and model predictions; r = ' + str(roun
d(r,3)))
```




```
correlation between human and model predictions; r = 0.691
```

Problem 2 (10 points)

In the cell below, briefly comment on whether or not the model is a good account of the human judgments. Which of the six scenarios do you think indicates that Player A is the strongest? Which of the scenarios indicates the Player A is the weakest? Does the model agree? Your reponse should be one or two paragraphs.

YOUR RESPONSE HERE

This model in general is good account of the human judgments because after the six scenarios, human conginition of player A strength and machine's strength level matched very closely. However, the model performed poorly in the early scenarios such as 1 and 2, which may be due to not having enough information/ having more uncertainty in the early stages of scenarios. Out of the six scenarios, I thought scenario 6 was the strongest indicator of player A being the strongest. For scenario 4, I thought it indicated Player A weakest.

The model agrees with my intution with scenario 6 as the indicator of Player A being strongest. This makes sense because Player A was the only undefeated player until 6th scenario, and scenario 5 really increased the probability of Player A being the strongest by adding new players into the match against player A. In addition, the model also agreed with my scenario 4 indicating Player A the weakest. This was mostly due to uncertainty of Player A's strength against Player B, who did very well against other players in scenario 4. As a result, both the model and I were not able to clearly tell whether Player A was truly the strongest player.

Problem 3 (15 points)

In the last problem, your job is to modify the probabilistic program to make the scenario slightly more complex. We have reimplemented the probabilistic program below with all the functions duplicated with a "_v2" flag.

The idea is that players may also have a "temper," which is a binary variable that is either `True` or `False`. Like `strength`, a player's temper is a PERSISENT variable that should be added to the world state. The probability that any given player has a temper is 0.2. Once a temper is sampled, its value persists until the world is cleared.

How does the temper variable change the model? If ALL the players on a team have a temper, the overall team strength (sum strength) is divided by 4! Otherwise, there is no effect.

Here is the assignment:

- First, write complete the function `has_temper` below such that each name is assigned a binary temper value that is persistent like strength. Store this temper value in the world state using `dict_temper.` [Hint: This function will look a lot like the `strength_v2` function]
- Second, modify the `team_strength_v2` function to account for the case that all team members have a temper.
- Third, run the simulation below comparing the case where Tom and Sue both have tempers to the case where Tom and Sue do not have tempers. How does this influence our inference about Bob's strength? Why? Write a one paragraph response in the very last cell explaining your answer.

In [11]:

```
class world_v2():
    def __init__(self):
        self.dict_strength = {}
        self.dict_temper = {}
    def clear(self): # used when sampling over possible world
        self.dict_strength = {}
        self.dict_temper = {}

def strength_v2(name):
    if name not in W.dict_strength:
        W.dict_strength[name] = abs(random.gauss(10,3))
    return W.dict_strength[name]

def lazy_v2(name):
    return random.random() < 0.1

def has_temper(name):
    # each player has a 0.2 probability of having a temper
    # TODO: YOUR CODE GOES HERE
    # pass # delete this line when done
    if name not in W.dict_temper:
        W.dict_temper[name] = random.random() < 0.2
    return W.dict_temper[name]

def team_strength_v2(team):
    # team : list of names
    mysum = 0.
    for name in team:
        if lazy_v2(name):
            mysum += (strength_v2(name) / 2.)
        else:
            mysum += strength_v2(name)
    # if all of the players have a temper, divide sum strength by 4
    ## TODO : YOUR CODE GOES HERE
    check_true = True
    for name in team:
        if has_temper(name) == False:
            check_true = False
    if check_true == True:
        mysum /= 4
    return mysum

def winner_v2(team1,team2):
    # team1 : list of names
    # team2 : list of names
    if team_strength_v2(team1) > team_strength_v2(team2):
        return team1
    else:
        return team2

def beat_v2(team1,team2):
    return winner_v2(team1,team2) == team1

W = world_v2()

f_return = lambda : strength_v2('bob')
```

```
list_f_conditions = [lambda : not has_temper('tom'), lambda : not has_te
mper('sue'), lambda : beat_v2( ['bob', 'mary'],['tom', 'sue'] ), lambda
: beat_v2( ['bob', 'sue'],  ['tom', 'jim'] )]
samples = rejection_sampler(f_return, list_f_conditions, nsamp=100000)
mean_strength = np.mean(samples)
print("If Tom and Sue do not have tempers...")
print("  Estimate of Bob's strength: mean = " + str(mean_strength) + ";
 effective n = " + str(len(samples)))

list_f_conditions = [lambda : has_temper('tom'), lambda : has_temper('su
e'), lambda : beat_v2( ['bob', 'mary'],['tom', 'sue'] ), lambda : beat_v
2( ['bob', 'sue'],  ['tom', 'jim'] )]
samples = rejection_sampler(f_return, list_f_conditions, nsamp=100000)
mean_strength = np.mean(samples)
print("If Tom and Sue BOTH have tempers...")
print("  Estimate of Bob's strength: mean = " + str(mean_strength) + ";
 effective n = " + str(len(samples)))
```

```
If Tom and Sue do not have tempers...
  Estimate of Bob's strength: mean = 11.814868227024009; effective n =
17281
If Tom and Sue BOTH have tempers...
  Estimate of Bob's strength: mean = 10.663921360144775; effective n =
1928
```

YOUR SHORT ANSWER GOES HERE. Does conditioning on temper influence our inference about Bob's strength?

Conditioning on temper influences our inference about Bob's strength. This is because when all the players in the opponent group have temper, their overall strength decreases 4 times. This means, the actual strength of Bob is more uncertain. In other words, we lose confidence in Bob's strength, and this is reflected in the decreased estimate of Bob's strength when both Sue and Tom have tempers.

In [ ]:

Homework - Bayesian modeling - Part C (30 points)

Implementing the Metropolis–Hastings algorithm for a Bayesian model of speech perception

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

This homework is due before midnight on Monday, April 4.

In this assignment, we examine a Bayesian model of speech perception and implement an approximate inference algorithm. As discussed in lecture, a "speaker" produces a speech sound T (e.g., a vowel sound) intended for a "listener". Because of noise during tranmission, the listener doesn't hear T exactly; instead, he hears the corrupted physical stimulus S.

The model postulates that the listener does a type of active reconstruction. Instead of verbatim perception of S, the listener aims to reconstruct the intended utterance T. Concretely, his/her perceptual system estimates $P(T|S)$, and this reconstructed stimulus is what they actually "hear." Reconstructing the details of the intended production is a good idea for a listener, since these details can be important for understanding what was said due to co-articulation.

This Bayesian model aims to explain the "perceptual magnet effect" in speech perception. This effect describes a particular kind of warping due to categorical representations. The phenomenon is that the perceived sound is closer to the category center than the raw stimulus S, in a way likened to a "perceptual magnet." In Bayesian terms, we will model this as computing the reconstruction as the expected value of $P(T|S)$ (which is denoted $E[T|S]$). See the figure below for an example.

We strongly suggest you read the David MacKay chapter (in NYU Classes resources), especially the section on Metropolis-Hastings, before proceedings with this assignment:

- MacKay, D. (2003). Chapter 29: Monte Carlo Methods. In Information Theory, Inference, and Learning Algorithms.

The Bayesian model of the perceptual margnet effect was introduced in this paper:

- Feldman, N. H., & Griffiths, T. L. (2007). A rational account of the perceptual magnet effect. In Proceedings of the Annual Meeting of the Cognitive Science Society. (http://ling.umd.edu/~nhf/papers/PerceptualMagnet.pdf)

In [1]:
```
# Import the necessary packages
import numpy as np
import random
from scipy.stats import norm
from scipy.special import logsumexp
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

Probabilistic model

Let's dive into the model details. First, we will need to set some key parameters.

In [2]:
```
# Key parameters we need for the probabilistic model
mu_c = 0 # category mean
sigma_c = 0.5 # category standard deviation
sigma2_c = sigma_c**2 # category variance
sigma_s = 0.4 # perceptual noise standard deviation
sigma2_s = sigma_s**2 # perceptual noise variance
X = np.linspace(-0.5,2,num=20) # points we are going to evaluate for war
ping
```

Prior

Lets start with the prior. This is the distribution of utterances a speaker says when producing a particular speech sound (e.g., a specific vowel). The prior distribution $P(T)$ on speaker productions T is modeled as a normal distribution

$$P(T) = N(\mu_c, \sigma_c^2).$$

This distribution is implemented in the `logprior_normal` function, which computes the log-probability. Although not necessary for this very simple case, it's important to ALWAYS compute with log-probabilities to prevent numerical underflow errors.

In [3]:
```
def logprior_normal(T):
    # Log-probability of speech production T
    return norm.logpdf(T,mu_c,sigma_c)
```

Likelihood

The production T is perturbed by noise to become the listener's perceived stimulus S. This noise process is also modeled as a normal distribution

$$P(S|T) = N(T, \sigma_S^2),$$

with the amount of noise governed by the standard deviation parameter σ_S. This distribution is implemented in `loglikelihood_normal`.

In [4]:
```
def loglikelihood_normal(S,T):
    # Log-probability of a stimulus S given production T
    return norm.logpdf(S,T,sigma_s)
```

Posterior mean, for model with normal prior and normal likelihood

In this Bayesian model, we assume that the goal of the listener is to optimally infer the intended production T given the perceived stimulus S. In other words, the listener is computing the posterior

$$P(T|S) = \frac{P(S|T)P(T)}{P(S)}.$$

As the prior and likelihood are normally distributed, we have a "conjugate prior", meaning the posterior takes the same distributional form as the prior. Thus $P(T|S)$ is also normally distributed. If you work out the math (a great exercise for those interested!), the posterior is

$$P(T|S) = N(\frac{\sigma_c^2 S + \sigma_S^2 \mu_c}{\sigma_c^2 + \sigma_S^2}, \frac{\sigma_c^2 \sigma_S^2}{\sigma_c^2 + \sigma_S^2}).$$

For the purposes of this assignment, we are only interested in the expected value (mean) of the posterior distribution, which is

$$E[T|S] = \frac{\sigma_c^2 S + \sigma_S^2 \mu_c}{\sigma_c^2 + \sigma_S^2},$$

corresponding to the "best guess" of the unobservable intended production T.

Notice that this is a weighted average between the actual stimulus S and the prior mean μ_c. The form of this average is intuitive. If the perceptual noise is high (high σ_S), the listener relies more on her prior expectations of about what the speech sound typically sounds like, giving μ_c a higher weight. If the prior expectation is highly variable (high σ_c), the listener relies more heavily on the perceived stimulus S. These are predictions the model makes, which have been empirically verified.

We provide the function `post_mean_normal_normal` for computing this "best guess" expected value of the posterior. This function is only valid when both, prior and likelihood, are each represented by a normal distribution.

In [5]:
```
def post_mean_normal_normal(S):
    # Posterior mean E[T|S] of sound production T given signal S, given
 normal P(T) and P(S|T)
    return (sigma2_c*S+sigma2_s*mu_c)/(sigma2_c+sigma2_s)
```

Visualizing the perceptual magnet effect

Let's see this Bayesian model in action. The `plot_warp` function visualizes how various raw stimulus values S warp to become perceived values $E[T|S]$. For its arguments, the parameter `stimuli_eval` is a numpy array of all of the values of S we want to evaluate. The function handle `f_posterior_mean` (see description below for a reminder about function handles) computes/estimates the posterior mean for a given stimulus S. The function handle `f_logprior` returns the log-probability of the prior for utterances T.

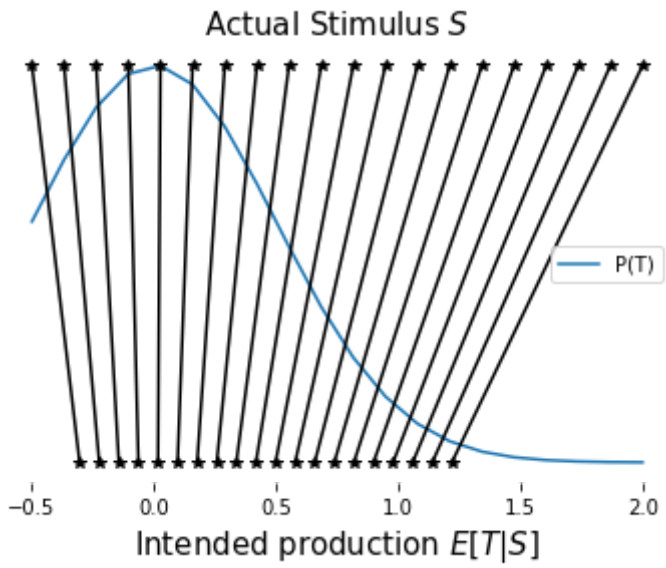
Let's run the `plot_warp` function for the normal-normal model that we have developed so far. The `post_mean_normal_normal` is the function handle for the posterior mean, and the function `logprior_normal` is the handle for the log prior.

This code produces a plot showing actual stimuli (S) at the top and perceived stimuli $E[T|S]$ at the bottom, overlaid with the category density $P(T)$ shown in blue. There is a clear "perceptual magnet effect" where perception warps the stimuli toward the category center.

Note: Function handles are pointers to a function which can be executed with the syntax `f_posterior_mean(S)` or `f_logprior(T)`. We need to pass handles, rather than pre-executed functions, so the script can control for itself when to execute the functions.

In [6]:
```
def plot_warp(f_posterior_mean,stimuli_eval,f_logprior,verbose=False):
    # Input
    #   f_posterior_mean : function handle f(S) that estimates posterior
mean E[T|S] for a raw stimulus S
    #      (only works for scalar S)
    #   stimuli_eval : [numpy array] of raw stimuli S we want to evalute
    #   f_logprior : function handle f(T) that evaluates log-prior log P
(T) for production T
    plt.figure()
    mypdf = np.exp(f_logprior(stimuli_eval))
    mx = np.max(mypdf)
    plt.plot(stimuli_eval,mypdf)
    for idx,x in enumerate(stimuli_eval):
        if verbose:
            print('  Estimating ' + str(idx+1) + ' of ' + str(len(stimul
i_eval)) + ' stimuli S')
        x_new = f_posterior_mean(x)
        plt.plot([x,x_new],[mx,0.],'k*-')
    plt.legend(['P(T)'])
    plt.tick_params(top=False, bottom=True, left=False, right=False, lab
elleft=False, labelbottom=True)
    for spine in plt.gca().spines.values():
        spine.set_visible(False)
    plt.title('Actual Stimulus $S$',size=15)
    plt.xlabel('Intended production $E[T|S]$',size=15)

print('Normal-normal model with exact inference')
plot_warp(post_mean_normal_normal,X,logprior_normal)
```

```
Normal-normal model with exact inference
```



Approximate inference with Metropolis-Hastings algorithm

So far, we have developed a simple normal-normal model, where the posterior mean can be computed in closed form, e.g., via `post_mean_normal_normal`. Usually, we are not so lucky, and a closed form solution is not available. In most cases, approximate inference algorithms are needed. Here, you will implement the very general and powerful Metropolis-Hasting algorithm for approximate inference using Markov Chain Monte Carlo (MCMC). See your lecture slides for the algorithm specification.

Metropolis-Hastings (MH) constructs a sequence of samples that converges to the posterior $P(T|S)$, if the sequence is run for long enough. At each step, a new value of T is proposed, and it is accepted or rejected based on its score using the MH "acceptance rule." Either way, the value of T is stored as a sample, and another proposal is made at the next step. A certain number of samples is thrown away at the beginning of the chain (burn in), and the remaining can be used to approximate the posterior $P(T|S)$. In this case, we want to estimate the posterior mean $E[T|S]$, so we average the samples with `np.mean(samples[nburn_in:])`.

Problem 1 (20 points)

Fill in the missing code below for a MH sampler for estimating E[T|S].

- There should be produces `nsamp` samples total, but with `nburn_in` samples thrown out from the beginning of the sequence.
- New proposals are made from a normal distribution centered at the current value of `T` with standard deviation `prop_width`.

More information on Metropolis-Hastings can be found in the David MacKay chapter on Monte Carlo methods in the Resources folder on NYU classes.

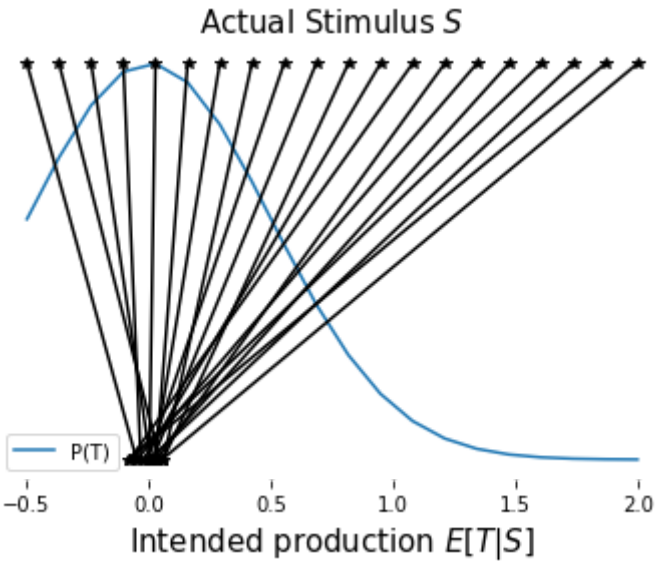
Hint: Computing the acceptance ratio a (see lecture slides) does not need to involve the normalizing constant $P(S)$ in the posterior,

$$P(T|S) = \frac{P(S|T)P(T)}{P(S)}.$$

This constant does not depend on the the variable T, the variable we are sampling, and it cancels out in the numerator and denominator when computing the ratio a. Thus it does not need to be included. This is critical since for many probabilistic models, we don't know the normalizing constant and thus can't use it in the acceptance ratio. This is one reason why MCMC is such a useful tool for probabilistic inference.

In [9]:
```
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=20(
,nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



Problem 2 (5 points)

Run your code from Problem 1 and examine the plot. Note that for each possible stimulus S, we run a different MCMC chain to estimate E[T|S].

- What is your reaction to the plot? How do the approximate estimates of E[T|S] using the sampler, compare to exact inference?
- What happens when you decrease the number of samples used in the MH algorithm?

YOUR RESPONSE GOES HERE

My reaction to the plot is that these graphs clearly show the perceptual magnet effect that occurs due to warping from categorical representations. This is shown on all plots where Stimulus S is mapped around 0 center. Compared to exact inference, MH algorithm's estimation of E[T|S] performs better even at lower sample size of 150.

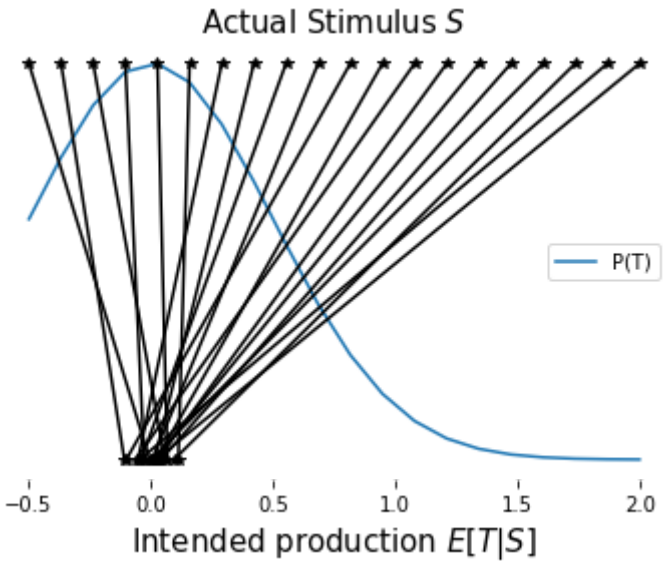
When I decrease the number of samples used in MH algorithm, perceived sound is not as concentrated on the category center as when larger samples are used. Due to the magnet effect, E[T|S] is mapped around 0.0 value (category center) more precisely at larger samples sizes. This is expected because more samples will create and demonstrate perceptual magnet effect much more clearly in a larger quantities.

Below graphs are created using different nsamp sizes at {1000, 500, 150}

In [18]:

```
# nsamp = 1000
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=1000
,nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

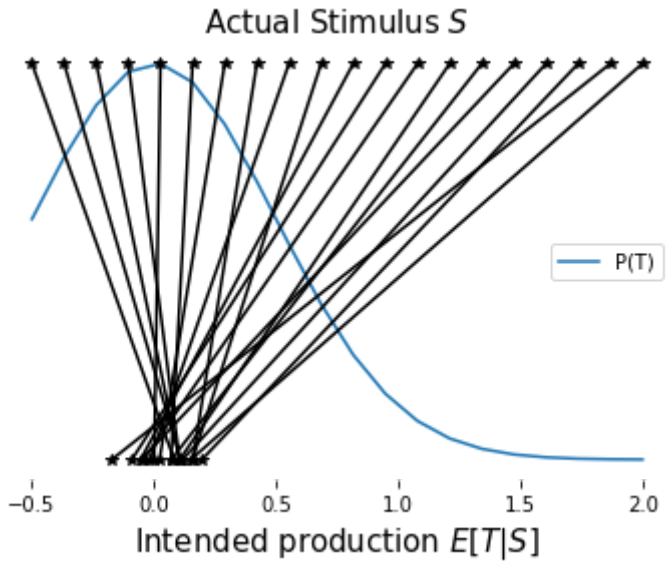
```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [16]:

```
# nsamp = 500
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=500,
nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

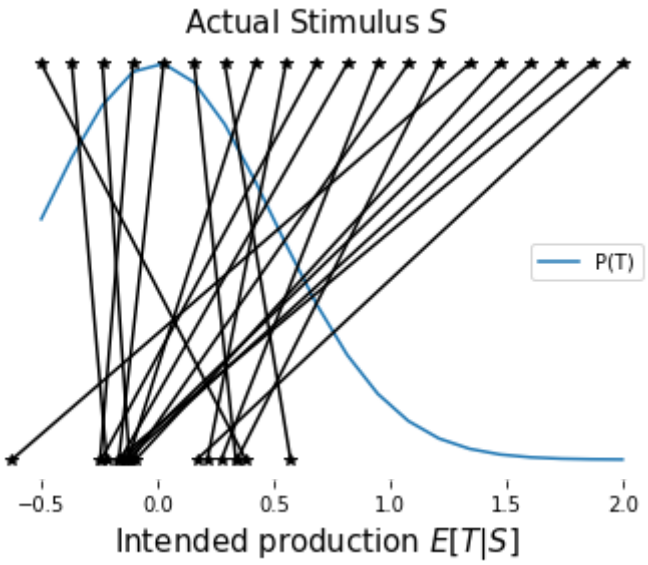
```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [17]:

```
# nsamp = 150
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=150,
nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



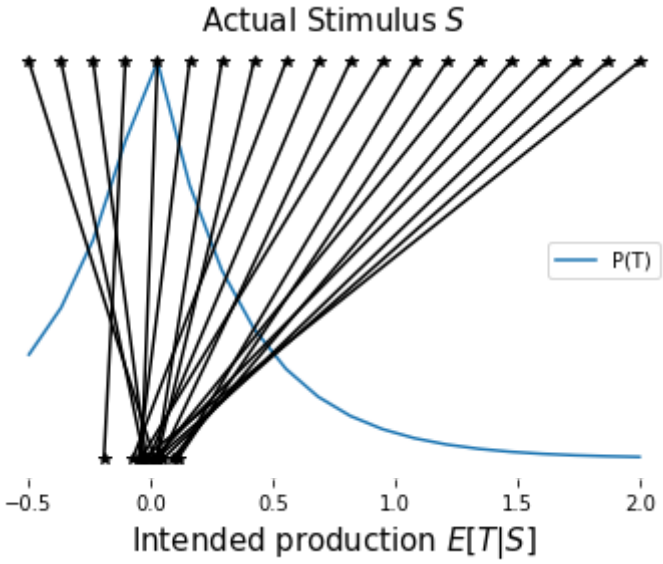
Non-conjugate Bayesian model of speech perception

To demonstrate the power and generality of the Metropolis-Hastings algorithm, let's change the probabilistic model. Rather than using a normal distribution $P(T)$ over speaker utterances, let's assume we have a Laplace distribution (https://en.wikipedia.org/wiki/Laplace_distribution) instead. It is unimodal like a normal distribution, but with fatter tails.

Use the code below to make a new plot. This time, we use the Laplace prior `logprior_laplace` over speaker utterances instead of the normal prior over utterances.

In [19]:
```
def logprior_laplace(T):
    # Alternative prior distribution
    #  log P(T|c) ~ Laplace(mu_c,b)
    b = sigma_c/np.sqrt(2)
    return (-np.abs(T-mu_c)/b) - np.log(2*b)

print('Laplace-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



Problem 3 (5 points)

- What effect did replacing the prior have on the model?
- Is there a perceptual magnet effect with the Laplace prior? Does the Bayesian explanation of the phenomenon depend on having a normal prior?

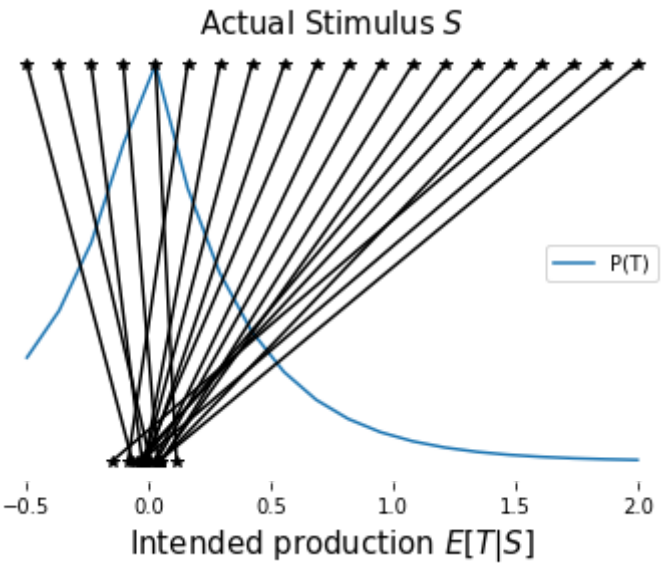
YOUR RESPONSE HERE

Replacing the prior using Laplace from normal had a small improvement, but it is difficult to say it made a siginificant difference. The accuracy was very similar in both Laplace and normal. This is because at larger sample sizes, both algorithms work very well, and it is only at the 150 sample size that we perceive less outliers away from the 0 center.

Perceptual magenet effect was also present with clear visualization in the Laplace prior as well. This means that Bayesian explanation of the phenomenon doesn't depend on having a normal prior, and also the accuracy stays similar for the most part.

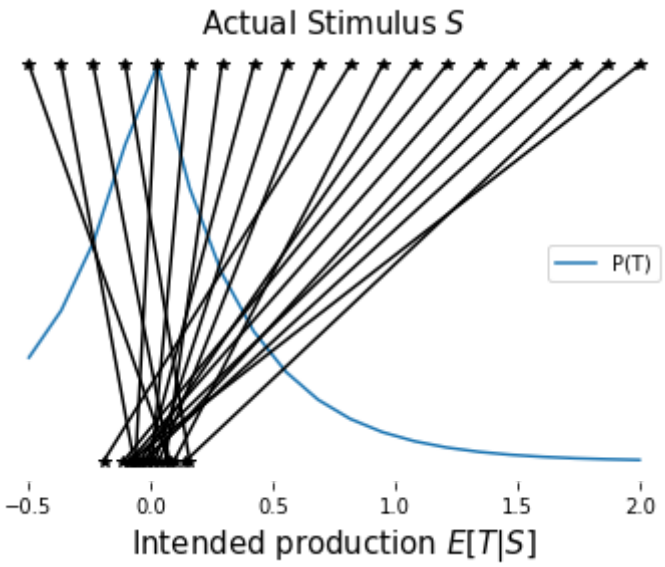
In [20]:
```
print('Laplace-normal model with MCMC inference @ nsamp=2000')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 2000)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=2000
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



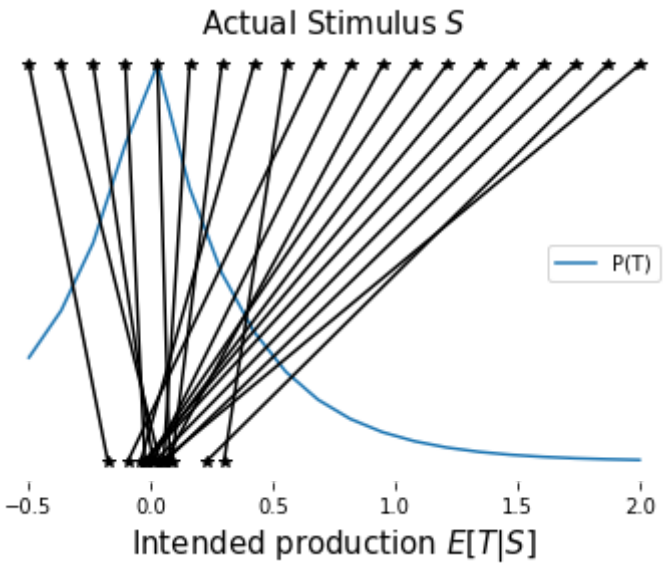
In [21]:
```
print('Laplace-normal model with MCMC inference @ nsamp=1000')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 1000)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=1000
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



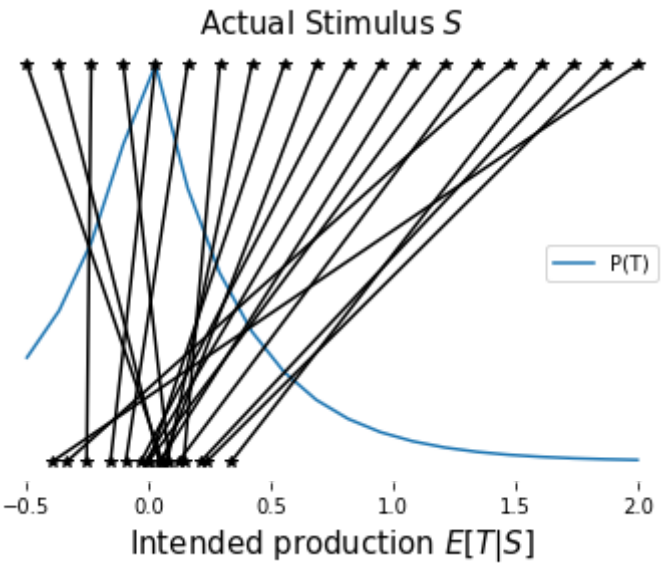
In [22]:
```
print('Laplace-normal model with MCMC inference @ nsamp=500')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 500)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=500
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [23]:
```
print('Laplace-normal model with MCMC inference @ nsamp=150')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 150)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=150
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [ ]: