Homework-Bayes-A_Answer

March 28, 2024

1 Homework - Bayesian modeling - Part A (100 points)

1.1 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).

1.1.1 The Bayesian model

In the number game, we receive a set of n positive examples $X = \{x^{(1)}, ..., x^{(n)}\}$ 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.

[1]:
```
# 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
x_max = 100 # (numbers between 1 and 100 are allowed)
```

[2]:
```
# 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,␣
 ↪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, 38, 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]
```

[3]:
```
## 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]
```

1.2 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.

[4]:
```
# The 30 test numbers that Tenenbaum used are here
x_eval =␣
 ↪[2,4,6,8,9,10]+list(range(12,23))+[24,25,26,28,32,36,41,56,62,64,87,95,96]
```

[5]:
```
x_eval
```

[5]:
```
[2,
 4,
 6,
 8,
 9,
 10,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 24,
 25,
 26,
 28,
 32,
 36,
 41,
 56,
 62,
 64,
 87,
 95,
 96]
```

1.3 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.

[6]:
```
#First, we declare the necessary variables
hypotheses = h_set_math + h_set_int
mylambda = 2./3
X1 = [16]
X2 = [16, 8, 2, 64]
X3 = [16, 23, 19, 20]
X_sets = [X1, X2, X3]
```

1.3.1 Log-Priors Function

For a hypothesis (h) and a set of observed data (X), the log-prior function is defined as:

$$\log \mathrm{P}(h) = \begin{cases} \log\left(\frac{\lambda}{|h|}\right) & \text{if h is a Math-Based Rule} \\ \log\left((1-\lambda)\frac{(\frac{|h|}{\sigma^2} exp - \frac{|h|}{\sigma})}{|h|}\right) & \text{if h is an Interval-Based Rule} \end{cases}$$

Where: - (n) is the number of observations in (X). - (|h|) is the cardinality of the hypothesis set (h) (i.e., the number of elements it includes).

The function returns (-∞) (representing a log probability of 0) if any element of (X) is not contained in (h), indicating that the hypothesis does not explain the data.

[7]:
```
def log_prior(h, mylambda=mylambda, n_math=len(h_set_math),␣
 ↪n_intervals=len(h_set_int), sigma=10):
    if h in h_set_math:
        return np.log(mylambda/n_math)
    else:
        card_h = len(h)
        erlang_prob = pdf_erlang(card_h, sigma=sigma)
        return np.log((1-mylambda) * erlang_prob/n_intervals)
```

1.3.2 Log-Likelihood Function

For a hypothesis (h) and a set of observed data (X), the log-likelihood function is defined as:

$$\log \mathrm{P}(X|h) = \begin{cases} n \cdot \log\left(\frac{1}{|h|}\right) & \text{if all } x \in X \text{ are contained in } h, \\ -\infty & \text{otherwise.} \end{cases}$$

Where: - (n) is the number of observations in (X). - (|h|) is the cardinality of the hypothesis set (h) (i.e., the number of elements it includes).

The function returns (-∞) (representing a log probability of 0) if any element of (X) is not contained in (h), indicating that the hypothesis does not explain the data.

[8]:
```
def log_likelihood(X, h):
    if all(x in h for x in X):
        return len(X) * np.log(1.0/len(h))
    else:
        return -np.inf
```

1.3.3 Log-Posterior Function

Given a set of hypotheses (H), their log prior probabilities, and a set of observed data (X), the log-posterior function computes the log-posterior probability for each hypothesis as follows:

$$\log \mathrm{P}(h|X) = (\log \mathrm{P}(X|h) + \log \mathrm{P}(h)) - \log \mathrm{sumexp}(\log \mathrm{P}(X|h) + \log \mathrm{P}(h))$$

This function calculates an array of log likelihoods for each hypothesis given (X), adds the log priors to these log likelihoods, and then normalizes the results by subtracting the logsumexp of the unnormalized log posteriors.

[9]:
```
def log_posterior(hypotheses, X):
    log_likelihoods = np.array([log_likelihood(X, h) for h in hypotheses])

    log_priors = np.array([log_prior(h) for h in hypotheses])

    log_posteriors = (log_likelihoods + log_priors) - logsumexp(log_likelihoods␣
↪+ log_priors)

    return log_posteriors
```

[10]:
```
log_posteriors = log_posterior(hypotheses, X1)
```

1.3.4 Predict Number Function

For a number (y), a set of hypotheses (H), and their log-posterior probabilities, the `predict_number` function calculates the probability that (y) is part of the concept:

$$\log P(y \in C \mid X) = \sum_{h \in H} (\log P(y \in C \mid h) + \log P(h|X)), \tag{4}$$

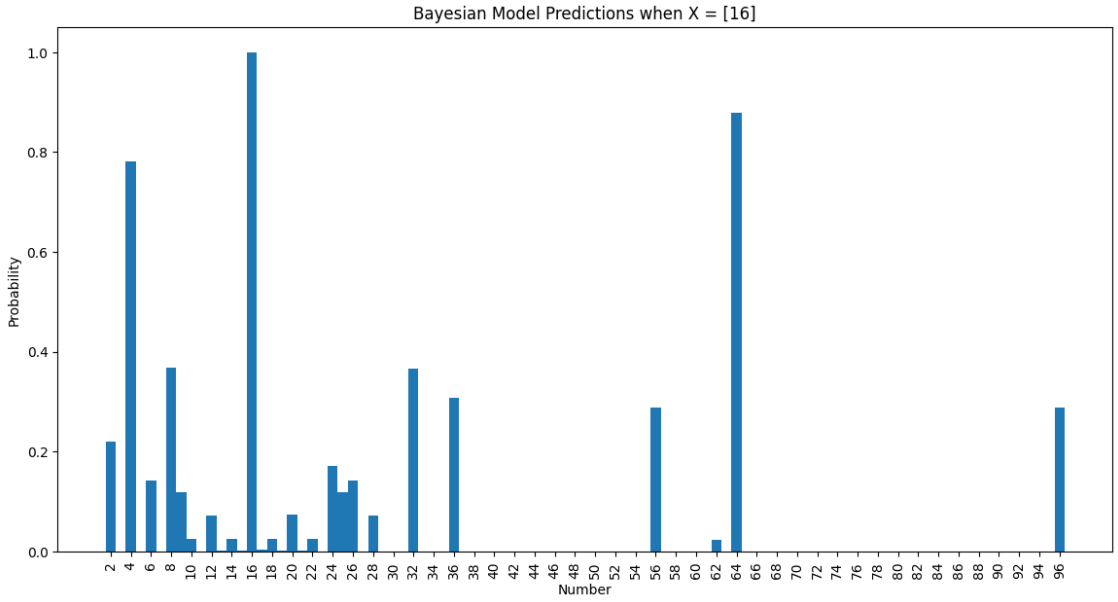
The function initializes the log probability that (y) is in the concept as (-∞) (log(0)). It iterates over all hypotheses, and if (y) is in a hypothesis, it updates the log probability using `np.logaddexp` with the corresponding log-posterior. The result is then exponent

[11]:
```
def y_predict(y, hypotheses, log_posteriors):
    log_prob_y = -np.inf

    for i,h in enumerate(hypotheses):
        if y in h:
            log_prob_y = np.logaddexp(log_prob_y, log_posteriors[i])

    return np.exp(log_prob_y)
```

[12]:
```
#Code to create plots
def plot_predictions(x_eval, predictions, X):
    plt.figure(figsize=(14, 7))
    plt.bar(x_eval, predictions, width=1.0)
    plt.xlabel('Number')
    plt.ylabel('Probability')
    plt.title(f'Bayesian Model Predictions when X = {X}')
    plt.xticks(range(min(x_eval), max(x_eval) + 1, 2), rotation=90)
    plt.show()
```

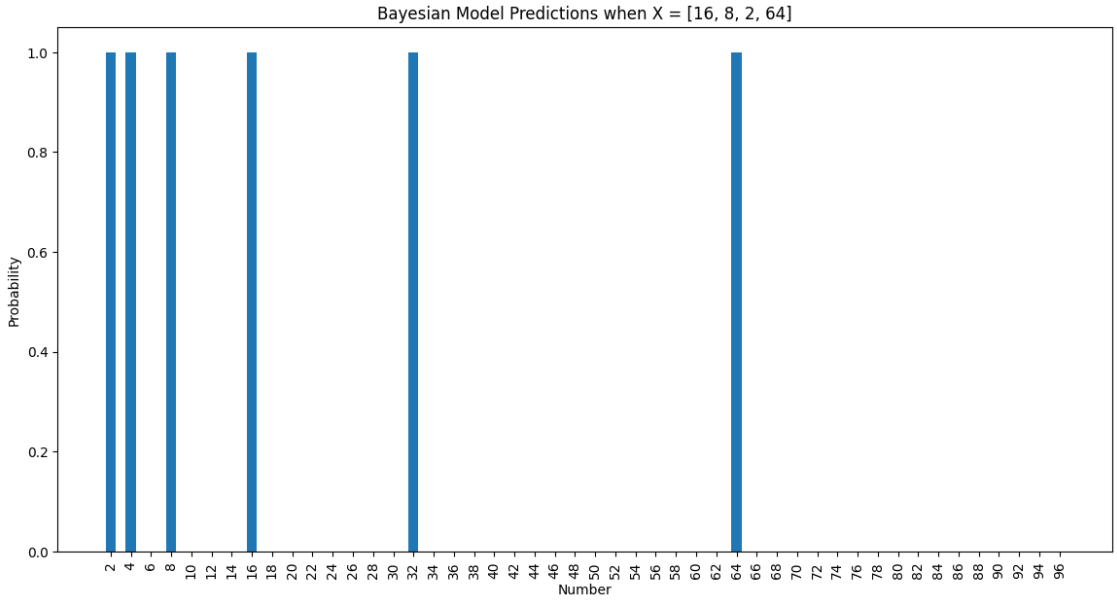
[15]:
```
def predict_and_plot(hypotheses, X_sets, x_eval):
    for X in X_sets:
        #Compute log posteriors for current set X
        log_posteriors = log_posterior(hypotheses, X)

        #Predict probabilities for all numbers in x_eval
        predicted_probabilities = [y_predict(y, hypotheses, log_posteriors) for␣
 ↪y in x_eval]

        #Plot predictions
        plot_predictions(x_eval, predicted_probabilities, X)

         #Returns the actual probability values
        for y in x_eval:
            prob = y_predict(y, hypotheses, log_posteriors)
            predicted_probabilities.append(prob)
            print(f"Predicted probability for the number {y}: {prob}")
```

[17]:
```
predict_and_plot(hypotheses, X_sets, x_eval)
```



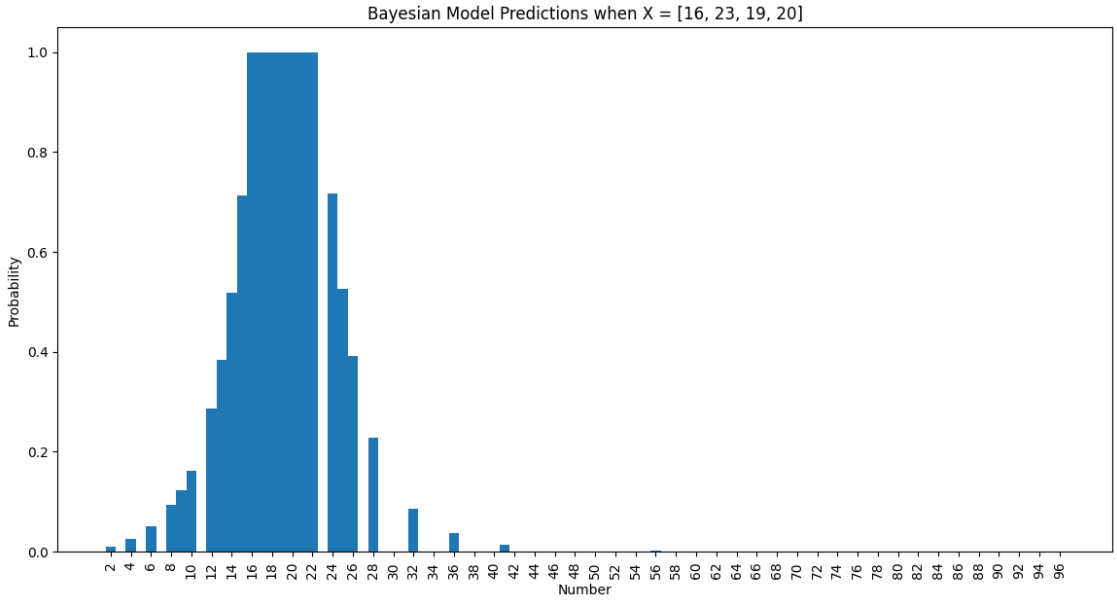

```
Predicted probability for the number 2: 0.2208193638459673
Predicted probability for the number 4: 0.7805206516943678
Predicted probability for the number 6: 0.14250085046935887
Predicted probability for the number 8: 0.3674063189089204
Predicted probability for the number 9: 0.11937226525397739
Predicted probability for the number 10: 0.025019139190964806
Predicted probability for the number 12: 0.07278248575153712
Predicted probability for the number 13: 0.0021421690986009718
Predicted probability for the number 14: 0.0260920453082017
Predicted probability for the number 15: 0.0027941318130318897
Predicted probability for the number 16: 1.0000000000000002
Predicted probability for the number 17: 0.0028704404332028453
Predicted probability for the number 18: 0.026237400823222186
Predicted probability for the number 19: 0.0023500008276801446
Predicted probability for the number 20: 0.07304684829683492
Predicted probability for the number 21: 0.0019239009175189403
Predicted probability for the number 22: 0.02538093645178155
Predicted probability for the number 24: 0.1708463860405971
Predicted probability for the number 25: 0.11949031826208356
Predicted probability for the number 26: 0.143007732971182
Predicted probability for the number 28: 0.07187559696407397
Predicted probability for the number 32: 0.36706277572045665
Predicted probability for the number 36: 0.3077511233285889
Predicted probability for the number 41: 0.0002598134364321407
Predicted probability for the number 56: 0.28767966492971664
Predicted probability for the number 62: 0.023671429622774704
Predicted probability for the number 64: 0.8786521707648461
```

```
Predicted probability for the number 87: 1.9724781506907913e-06
Predicted probability for the number 95: 5.307709441739612e-07
Predicted probability for the number 96: 0.2876226130968137
```




```
Predicted probability for the number 2: 1.0000000000000004
Predicted probability for the number 4: 1.0000000000000004
Predicted probability for the number 6: 0.00020732146226298825
Predicted probability for the number 8: 1.0000000000000004
Predicted probability for the number 9: 4.452441401758737e-09
Predicted probability for the number 10: 0.00020732146226298825
Predicted probability for the number 12: 0.00020732146226298825
Predicted probability for the number 13: 4.452441401758737e-09
Predicted probability for the number 14: 0.00020732146226298825
Predicted probability for the number 15: 4.452441401758737e-09
Predicted probability for the number 16: 1.0000000000000004
Predicted probability for the number 17: 4.452441401758737e-09
Predicted probability for the number 18: 0.00020732146226298825
Predicted probability for the number 19: 4.452441401758737e-09
Predicted probability for the number 20: 0.00020732146226298825
Predicted probability for the number 21: 4.452441401758737e-09
Predicted probability for the number 22: 0.00020732146226298825
Predicted probability for the number 24: 0.00020732146226298825
Predicted probability for the number 25: 4.452441401758737e-09
Predicted probability for the number 26: 0.00020732146226298825
Predicted probability for the number 28: 0.00020732146226298825
Predicted probability for the number 32: 1.0000000000000004
Predicted probability for the number 36: 0.00020732146226298825
Predicted probability for the number 41: 4.452441401758737e-09
```

```
Predicted probability for the number 56: 0.00020732146226298825
Predicted probability for the number 62: 0.00020732146226298825
Predicted probability for the number 64: 1.0000000000000004
Predicted probability for the number 87: 1.602698450757805e-10
Predicted probability for the number 95: 3.6127503604417515e-11
Predicted probability for the number 96: 0.0002073170377967166
```




```
Predicted probability for the number 2: 0.01025400693298449
Predicted probability for the number 4: 0.026156910796873066
Predicted probability for the number 6: 0.0513967476238877
Predicted probability for the number 8: 0.0926058655144713
Predicted probability for the number 9: 0.12274545920649202
Predicted probability for the number 10: 0.16232653396153338
Predicted probability for the number 12: 0.28586560432309427
Predicted probability for the number 13: 0.3830657413625491
Predicted probability for the number 14: 0.5189152459987427
Predicted probability for the number 15: 0.7133575673382103
Predicted probability for the number 16: 0.9999999999999986
Predicted probability for the number 17: 0.9999999999999986
Predicted probability for the number 18: 0.9999999999999986
Predicted probability for the number 19: 0.9999999999999986
Predicted probability for the number 20: 0.9999999999999986
Predicted probability for the number 21: 0.9999999999999986
Predicted probability for the number 22: 0.9999999999999986
Predicted probability for the number 24: 0.7170720176453852
Predicted probability for the number 25: 0.5256547367420609
Predicted probability for the number 26: 0.39227837305464935
Predicted probability for the number 28: 0.2278523175319164
```

```
Predicted probability for the number 32: 0.0865850270415384
Predicted probability for the number 36: 0.03655088107508686
Predicted probability for the number 41: 0.013692366287173083
Predicted probability for the number 56: 0.0010392940700271168
Predicted probability for the number 62: 0.0004066621732929255
Predicted probability for the number 64: 0.0002997302701638046
Predicted probability for the number 87: 9.585422945121582e-06
Predicted probability for the number 95: 2.1139552724526747e-06
Predicted probability for the number 96: 1.633337556440365e-06
```

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.

I found the model to be quite convincing, and it honestly surprised me that it's possible to approximate the human judgments of the number game with something as simple as Bayes Theorem! While there are some clear differences between the human judgments and the model's predictions, it seems to capture the general trends that are observed in the human data, as it also uses prior knowledge about numbers and updates beliefs based on observed evidence. However, the most obvious difference I noticed in the model predictions were the confidence levels and lack of variability. For example, when X = [16, 8, 2, 64], the model converges on the "Powers of 2" hypothesis (due to the size principle), whereas the humans gave higher probability predictions for other even numbers. This is observed yet again when X = [16], as the humans gave 2 a predicted probability of ~0.55, and the model gave a prediction of ~0.22. Humans seem to be more lenient in their estimations, possibly considering a broader range of factors, and the model does not fully capture the more nuanced and contextually flexible ways of thinking.

When considering more naturalistic settings, Bayesian modeling seems to be highly relevant, as humans often learn concepts by generalizing from a few learned examples. This is also related to where the hypothesis space comes from. In naturalistic settings, hypotheses are formed from prior knowledge (such as schemas), which allow us to pull from past experiences to make judgments, and update this knowledge when something like a prediction error occurs. While us humans may not necessarily iterate through all of the different possible hypotheses in the same way that Bayesian models do, we instead rely on things like pattern recognition, schemas, analogies / metaphors, and contextual cues that guide our inferences in a way that is far less computationally intensive than brute iteration.

Homework-Bayes-B_Answer

March 28, 2024

1 Homework - Bayesian modeling - Part B (40 points)

1.1 Probabilistic programs for productive reasoning

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

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 specially designed probabilistic programming languages such as WebPPL. Other languages have been introduced that combine aspects of probabilistic programming and neural networks, such as Pyro, and Edward. 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)

1.2 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 affect 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.

1.2.1 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.

[5]:
```
# 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
```

1.2.2 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.

[6]:
```
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]
```

1.2.3 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.

[7]:
```
def lazy(name):
    return random.random() < 0.1
```

[8]:
```
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
```

1.2.4 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`).

[9]:
```
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
```

2 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.

[15]:
```
def rejection_sampler(f_return, list_f_conditions, nsamp=10000):
    # Input
    #  f_return : function handle that grabs the variable of interest when␣
 ↪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 actual,␣
 ↪accepted samples
    samples = []
    for i in range(nsamp):
        # TODO : your code goes here (don't forget to call W.clear() before␣
 ↪each attempted sample)
        W.clear()
        #Check for all conditions before storing anything
        if all(f() for f in list_f_conditions):
            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).

[16]:
```
f_return = lambda : strength('bob')
list_f_conditions = [lambda : beat( ['bob', 'mary'],['tom', 'sue'] ), lambda :␣
 ↪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) + "; effective␣
 ↪n = " + str(len(samples)))
```

```
Estimate of Bob's strength: mean = 11.870629285045252; effective n = 14125
```

2.1 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.

[49]:
```
# TODO : YOUR DATA GOES HERE
subject1_pred = np.array([7,5,7,4,6,7])
subject2_pred = np.array([6,6,7,5,7,7])
```

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`

[50]:
```
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(samples)))
model_pred.append(np.mean(samples))
```

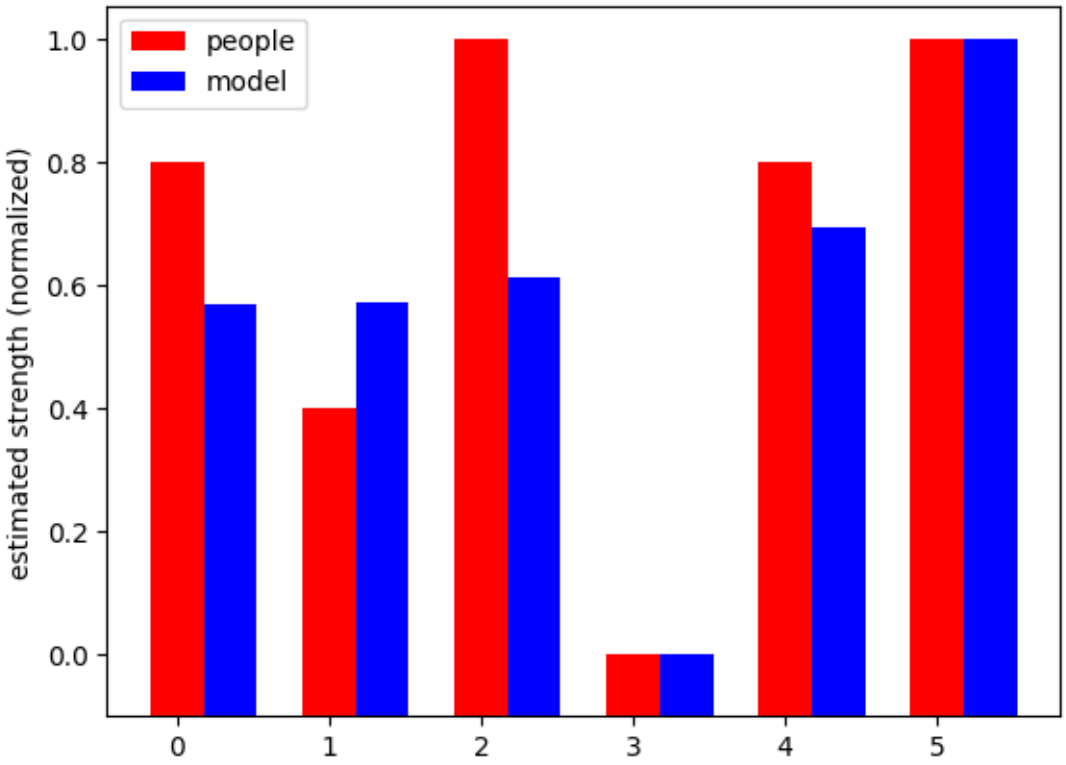
```
f_conditions = [lambda : beat( ['A', 'B'],['E', 'F'] ), lambda : beat( ['A',␣
 ↪'C'],['E', 'G'] ), lambda : beat( ['A', 'D'],  ['E', 'H'] ) ]
samples = rejection_sampler(f_return, f_conditions)
print("Scenario 2")
print("  sample mean : " + str(np.mean(samples)) + "; n=" + str(len(samples)))
model_pred.append(np.mean(samples))

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(samples)))
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(samples)))
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(samples)))
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(samples)))
model_pred.append(np.mean(samples))
```

```
Scenario 1
  sample mean : 12.078840545420062; n=2132
Scenario 2
  sample mean : 12.084466602532808; n=2130
Scenario 3
  sample mean : 12.200163834782861; n=773
Scenario 4
  sample mean : 10.519469505523324; n=2796
Scenario 5
  sample mean : 12.4241217027911; n=1715
Scenario 6
```

```
sample mean : 13.259137203863684; n=1288
```

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

[51]:
```
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(round(r,3)))
```




```
correlation between human and model predictions; r = 0.868
```

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.

Given the very high correlation between our human ratings and the output of the model, we can assume that this is indeed a good account of the human judgments. It seems as though scenario 6 indicates that Player A is the strongest, given the fact that Player A takes turns playing with and against Players B, C, and D, but still wins regardless of who they are playing with. In scenario 4, however, Player B beats Players E and F, regardless of whether they are playing with Player A or not, so it seems as though a high strength for Player A is not necessarily required in scenario 4. It is quite surprising and impressive how well the model is able to replicate human judgments.

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.

[52]:
```
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
    if name not in W.dict_temper:
        W.dict_temper[name] = random.random() < 0.2  # 20% chance of having a␣
 ↪temper
    return W.dict_temper[name]

def team_strength_v2(team):
    # team : list of names
    mysum = 0.
    all_have_temper = all(has_temper(name) for name in team)
    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
    if all_have_temper:
```

```
        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_temper('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('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 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 = 11.831297574454844; effective n = 17174
If Tom and Sue BOTH have tempers…
  Estimate of Bob's strength: mean = 10.775715798328457; effective n = 2045
```

The simulation results suggest that conditioning on temper greatly influences our inference about Bob's strength. When we know that neither Tom nor Sue have a temper, then Bob's estimated strength is 11.86 with a fairly large number of effective samples. However, when Tom and Sue both have tempers, Bob's strength drops to 10.81, and there are way less effective samples (1962 vs. 17,259 without tempers) to draw from. The decrease in Bob's estimated strength is likely due to the fact that when Tom and Sue both have temperaments, their team strength decreases by 75%, and the fact that Bob and Mary beat them becomes less indicative of Bob's overall skills and is more apparently a result of the weakened team he is up against. The decrease in the number

of effective samples is due to the fact that the combined probability of both Tom and Sue having tempers at the same time is actually 0.04 or 4% instead of the 20% chance that either one of them might have a temper. Thus, we end up with far fewer samples that meet the conditions in question due to rejection sampling. This also demonstrates the fact that as the conditions for accepting samples become more specific and less probable, the efficiency of the sampling process decreases.

Homework-Bayes-C_Answer

March 28, 2024

1 Homework - Bayesian modeling - Part C (30 points)

1.1 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/

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 class readings), especially the section on Metropolis-Hastings, before proceeding 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)

[11]:
```
# 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
```

1.2 Probabilistic model

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

[12]:
```
# 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 warping
```

1.2.1 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.

[13]:
```
def logprior_normal(T):
    # Log-probability of speech production T
    return norm.logpdf(T,mu_c,sigma_c)
```

1.2.2 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`.

[15]:
```
def loglikelihood_normal(S,T):
    # Log-probability of a stimulus S given production T
    return norm.logpdf(S,T,sigma_s)
```

1.2.3 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.

[16]:
```
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)
```

1.3 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.

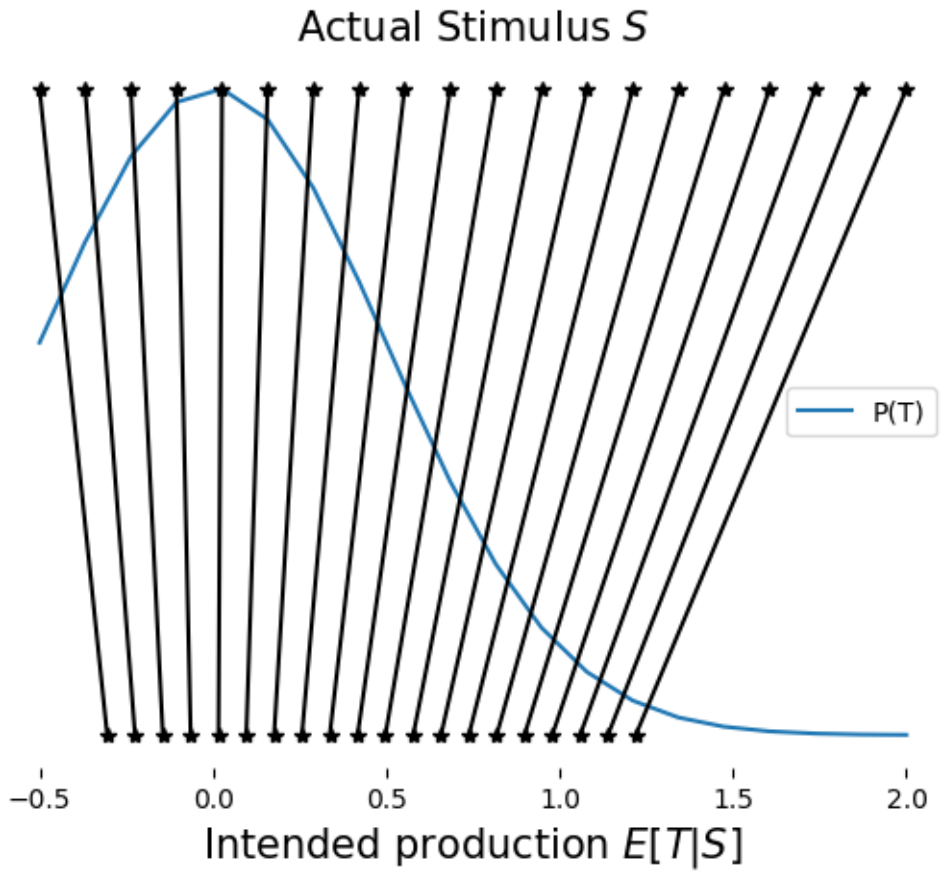
Let's run the `plot_warp` function for the normal-normal model that we have developed so far. The `post_mean_normal_normal` is the function handle for the posterior mean, and the function `logprior_normal` is the handle for the log prior.

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.

[17]:
```
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(stimuli_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,␣
 ↪labelleft=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
```



1.4 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.

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

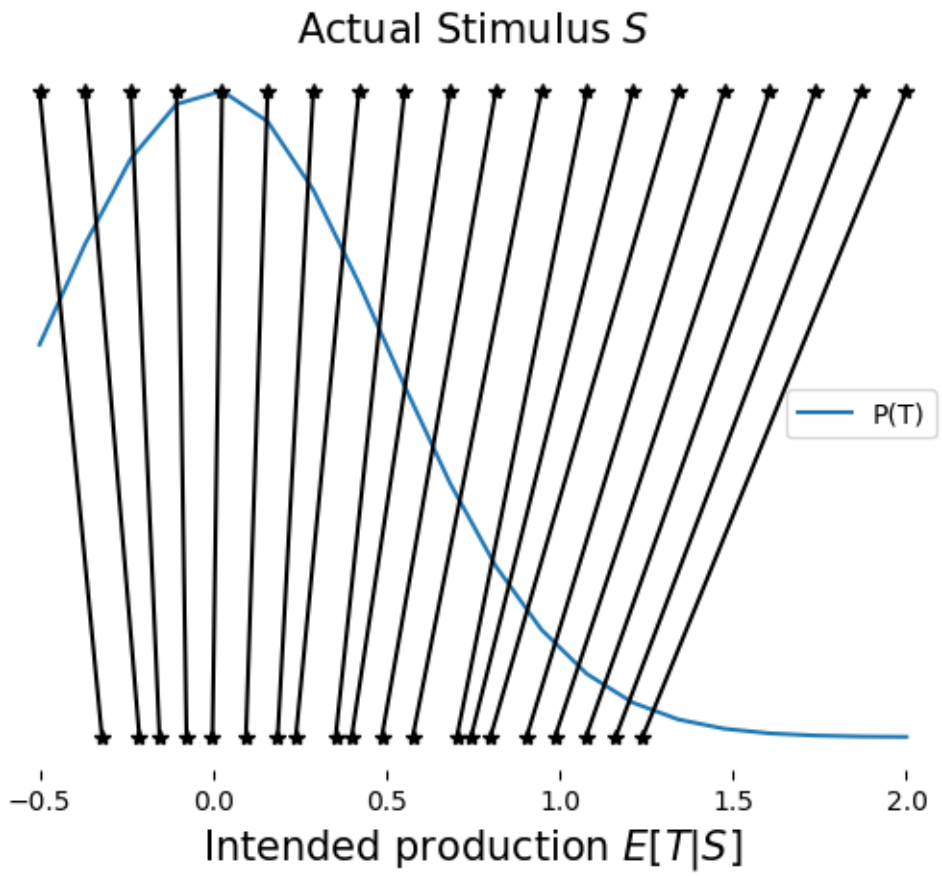
$$P(T|S) = \frac{P(S|T)P(T)}{P(S)}.$$

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.

[24]:
```
def␣
↪estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=2000,nburn_in=100,prop_widt
↪25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from the␣
↪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

    h_0 = np.random.randn() # Random initial sample h^(0)

    for t in range(nsamp):
        h_prime = h_0 + np.random.normal(0, prop_width)
        log_acc_ratio = (f_loglikelihood(S, h_prime) + f_logprior(h_prime)) -␣
↪(f_loglikelihood(S, h_0) + f_logprior(h_0))
        a = np.exp(log_acc_ratio)

        if a >= 1:
```

```
            h_0 = h_prime
        else:
            u = np.random.uniform(0, 1)
            if u < a:
                h_0 = h_prime

        samples.append(h_0)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S :␣
 ↪estimate_metropolis_hastings(S,loglikelihood_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)

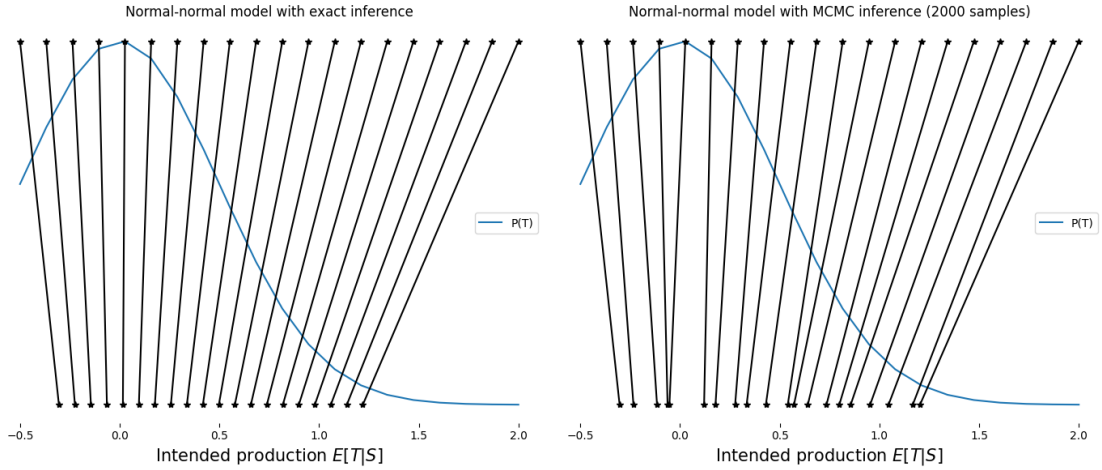
Run your code from Problem 1 and examine the plot. Note that for each possible stimulus S, we run a different MCMC chain to estimate E[T|S].

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?

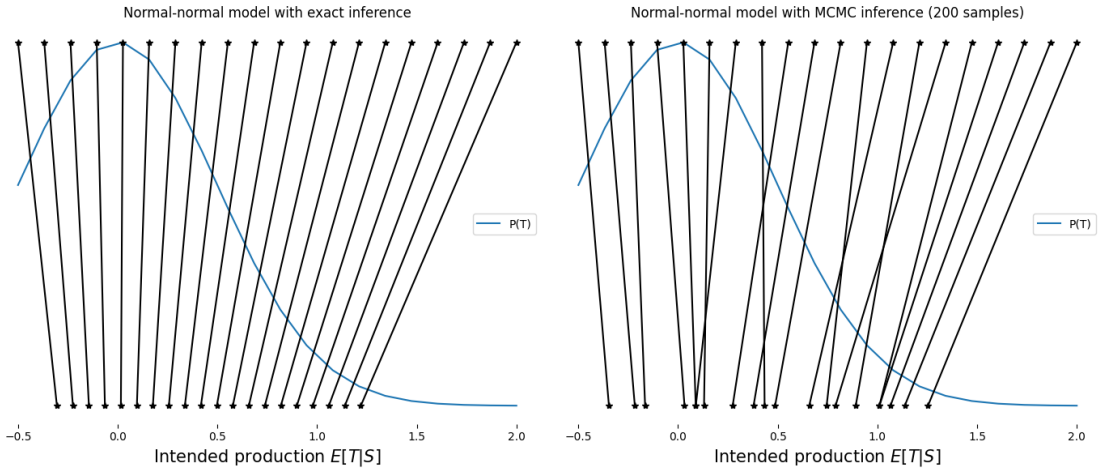
[32]:
```
def plot_warp_subplots(f_posterior_mean, stimuli_eval, f_logprior, ax,␣
↪verbose=False):
    # Modified plot_warp to accept an axis object 'ax' for plotting
    mypdf = np.exp(f_logprior(stimuli_eval))
    mx = np.max(mypdf)
    ax.plot(stimuli_eval, mypdf)
    for idx, x in enumerate(stimuli_eval):
        x_new = f_posterior_mean(x)
        ax.plot([x, x_new], [mx, 0.], 'k*-')
    ax.legend(['P(T)'])
```

```
    ax.tick_params(top=False, bottom=True, left=False, right=False,␣
 ↪labelleft=False, labelbottom=True)
    for spine in ax.spines.values():
        spine.set_visible(False)
    ax.set_xlabel('Intended production $E[T|S]$', size=15)

# Prepare the plot layout
fig, axs = plt.subplots(1, 2, figsize=(14, 6))

# Plot exact inference on the left
axs[0].set_title('Normal-normal model with exact inference')
plot_warp_subplots(post_mean_normal_normal, X, logprior_normal, axs[0])

# Plot Metropolis-Hastings inference on the right
axs[1].set_title('Normal-normal model with MCMC inference (2000 samples)')
f_posterior_mean = lambda S: estimate_metropolis_hastings(S,␣
 ↪loglikelihood_normal, logprior_normal)
plot_warp_subplots(f_posterior_mean, X, logprior_normal, axs[1], verbose=True)

# Show the complete plot with both subplots
plt.tight_layout()
plt.show()
```



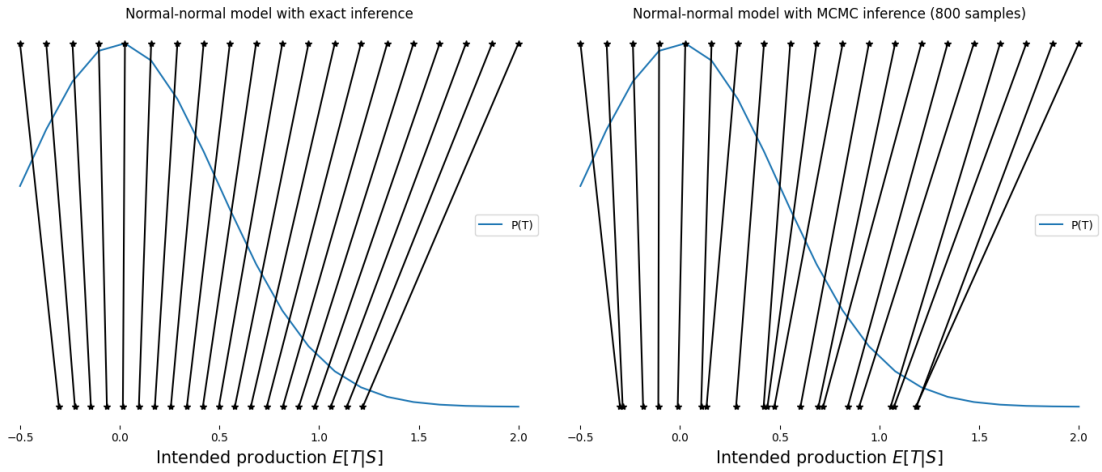
1.4.1 Here we see that the approximate estimates of E[T|S] using the sampler also demonstrate the "magnet effect" and align well with that of the exact inference plot. While there are subtle differences (which is to be expected given MCMC is an approximation method and can have some variance), this would likely be accounted for given more samples.

[35]:
```
# Prepare the plot layout
fig, axs = plt.subplots(1, 2, figsize=(14, 6))  # 1 row, 2 columns

# Plot exact inference on the left
axs[0].set_title('Normal-normal model with exact inference')
plot_warp_subplots(post_mean_normal_normal, X, logprior_normal, axs[0])

# Plot Metropolis-Hastings inference on the right
axs[1].set_title('Normal-normal model with MCMC inference (200 samples)')
f_posterior_mean = lambda S: estimate_metropolis_hastings(S,␣
 ↪loglikelihood_normal, logprior_normal, nsamp=200)
plot_warp_subplots(f_posterior_mean, X, logprior_normal, axs[1], verbose=True)

# Show the complete plot with both subplots
plt.tight_layout()
plt.show()
```



[36]:
```
# Prepare the plot layout
fig, axs = plt.subplots(1, 2, figsize=(14, 6))  # 1 row, 2 columns

# Plot exact inference on the left
axs[0].set_title('Normal-normal model with exact inference')
plot_warp_subplots(post_mean_normal_normal, X, logprior_normal, axs[0])

# Plot Metropolis-Hastings inference on the right
```

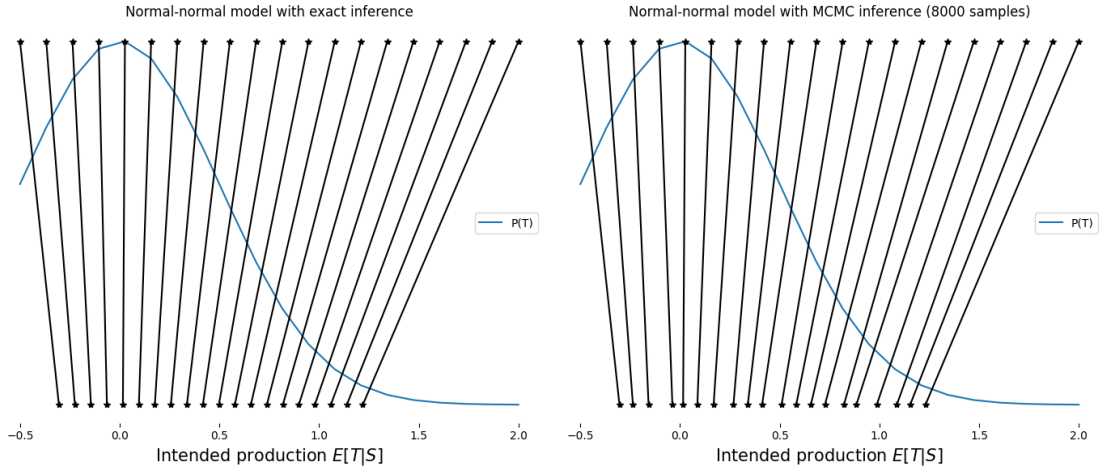
```
axs[1].set_title('Normal-normal model with MCMC inference (800 samples)')
f_posterior_mean = lambda S: estimate_metropolis_hastings(S,␣
 ↪loglikelihood_normal, logprior_normal, nsamp=800)
plot_warp_subplots(f_posterior_mean, X, logprior_normal, axs[1], verbose=True)

# Show the complete plot with both subplots
plt.tight_layout()
plt.show()
```



1.4.2 While the magnet effect is still observed with fewer samples (200 and 800), we also see a drastic increase in variability in the warp from the actual stimulus S to the intended production E[T|S], as the precision and confidence of the estimates are compromised. This could be due to the fact that there are not enough iterations for the Markov chain to fully converge in the way it does given more samples.

[37]:
```
# Prepare the plot layout
fig, axs = plt.subplots(1, 2, figsize=(14, 6))  # 1 row, 2 columns

# Plot exact inference on the left
axs[0].set_title('Normal-normal model with exact inference')
plot_warp_subplots(post_mean_normal_normal, X, logprior_normal, axs[0])

# Plot Metropolis-Hastings inference on the right
axs[1].set_title('Normal-normal model with MCMC inference (8000 samples)')
f_posterior_mean = lambda S: estimate_metropolis_hastings(S,␣
 ↪loglikelihood_normal, logprior_normal, nsamp=8000)
plot_warp_subplots(f_posterior_mean, X, logprior_normal, axs[1], verbose=True)

# Show the complete plot with both subplots
```

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



1.4.3 However, here we can see that as we increase the number of samples (8000), the MCMC inference is more in line with the exact inference.

1.5 Non-conjugate Bayesian model of speech perception

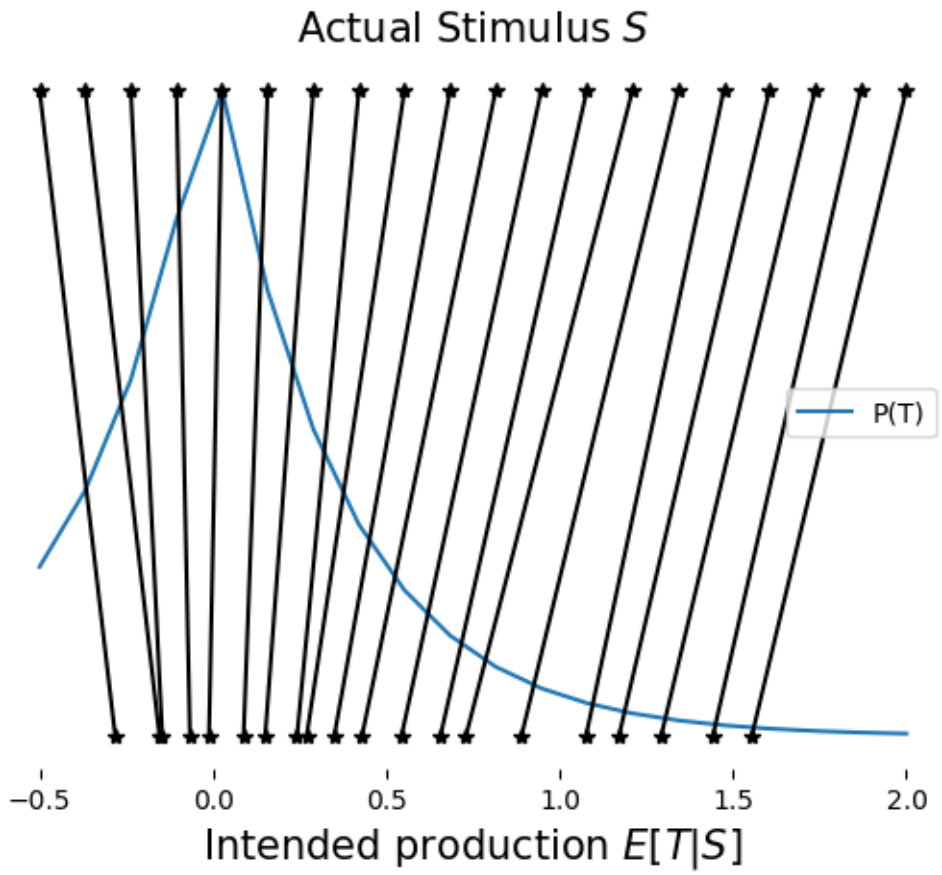
To demonstrate the power and generality of the Metropolis-Hastings algorithm, let's change the probabilistic model. Rather than using a normal distribution $P(T)$ over speaker utterances, let's assume we have a Laplace distribution instead. It is unimodal like a normal distribution, but with fatter tails.

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.

[20]:
```
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,loglikelihood_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?

Replacing the prior leads to a less pronounced perceptual "magnet effect" on the tails (e.g., when $S = 2.0$, it is merely pulled to ~1.55 instead of ~1.3 with the normal prior), and a more pronounced "magnet effect" near the mean. This makes sense, given the heavier tails of the Laplace distribution, which allow for more variation. While the Bayesian explanation of the phenomenon may still be demonstrated in cases of non-normal priors, such as the Laplace distribution exemplified above, utilizing a normal prior seems to be much more aligned with the exact perceptual inferences in this problem. A normal prior is likely the optimal choice in most perceptual models since perception is influenced by prior beliefs or knowledge about the stimuli.