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
```

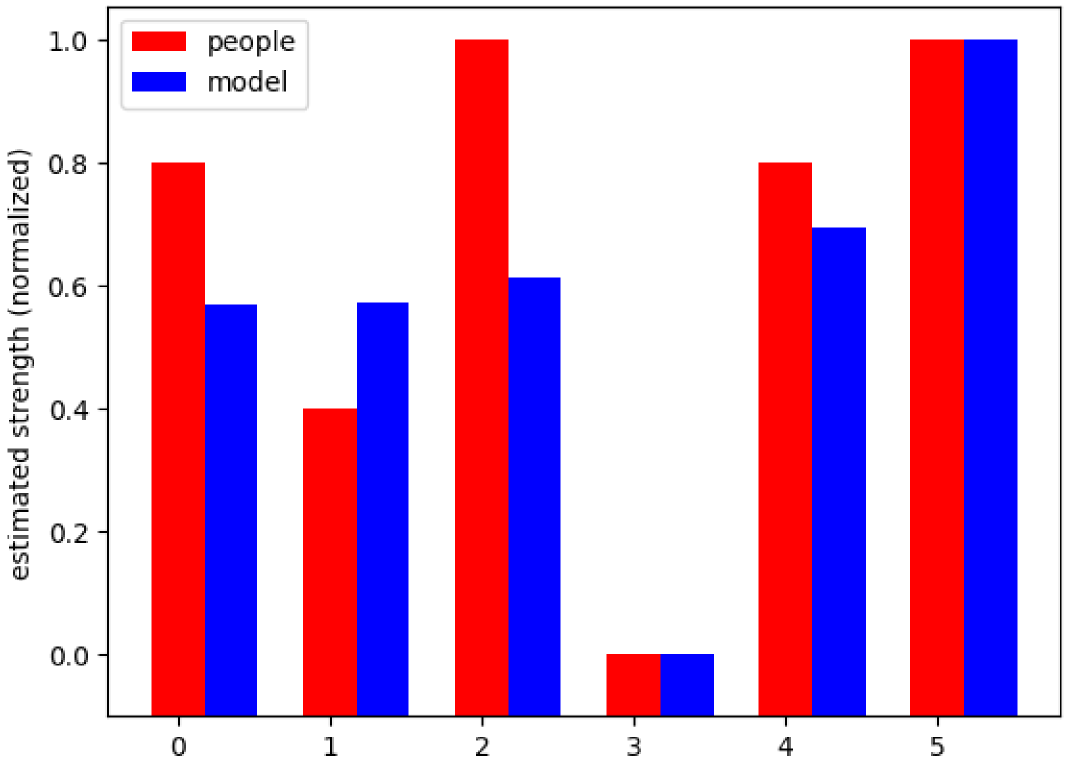
This code creates a bar graph to compare the human and model predictions for Player A's strength.

```
[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.