ISTA 311 Programming assignment 2: Bayesian Inference

Due: Tuesday, October 29, 11:59 PM

You are encouraged to collaborate with other students on this assignment. If you do so, please note the name of your collaborator in the header comment of your script.

1.1 Summary

Your script should satisfy the following parameters:

- The file should be called `inference.py`
- Include the following import: `from distribution import Distribution`
- The file should define four classes:
  - `InferenceSuite` (subclass of `Distribution`)
  - `Diagnostic` (subclass of `InferenceSuite`)
  - `Cookie` (subclass of `InferenceSuite`)
  - `Locomotive` (subclass of `InferenceSuite`)

A template including the method signatures for `InferenceSuite` and a fully worked example subclass is provided.

1.2 A class for Bayesian inference (20 points)

In the first programming assignment, we defined a class called `Distribution`, which is a wrapper for a so that the set of keys is the universe (sample space) and the corresponding values are the probabilities of the outcomes. In this assignment, we would like to expand this functionality to allow updating of the probabilities according to Bayes' theorem.

To do this, we will define a subclass of `Distribution` called `InferenceSuite`. This class models a collection of hypotheses (recall that we used the term *suite* to refer to a collection of hypotheses that is mutually exclusive and exhaustive) in the same way that the `Distribution` class modeled a collection of outcomes. That is, in the internal dictionary `self.d` of an `InferenceSuite`, the keys represent the hypotheses and the values represent their plausibilities (i.e. our current level of belief in each hypothesis).

To define the class as a subclass, use the line:

```
class InferenceSuite(Distribution)
```

Since `InferenceSuite` is a subclass of `Distribution`, it already has access to all of the methods we defined before: `prob`, `normalize`, `condition`, and `sample`. In addition to these, `InferenceSuite` will have the following methods:

- `update(self, data)`, which should take a single parameter which represents an observation or piece of evidence. `update` should iterate through the dictionary and update the probabilities using Bayes' theorem to incorporate the evidence. Note that the evidence does not necessarily have to be one of the outcomes from the dictionary. (Hint: iterate through the dictionary and update $P(H)$ to $P(E|H)P(H)$ (i.e., likelihood $\times$ prior) for each outcome in the dictionary, then call your `normalize` method.)

  In order for `update` to work correctly, it must call a likelihood function. However, since the likelihood function may be different for different situations, it is left unimplemented in the base `InferenceSuite` class. It will be implemented in subclasses.

- `map(self)`, which should return the hypothesis associated to the highest probability (the maximum a posteriori estimate). If there is a tie, return the first outcome that appears in the dictionary with that probability.
- `mean(self)`, which should return the mean of the random variable associated with the probability distribution. (Of course, this only makes sense if the outcomes are numerical. You're not required to do any type checking or error handling for this, but you can if you want.)
- `quantile(self, p)`, which should take a single parameter representing a probability p, and return the smallest hypothesis value x such that the cumulative probability for values less than x is at least p. (In other words, `self.quantile(0.90)` should return the 90th percentile value, etc.)
- **Do not implement** `likelihood`, but for the purposes of writing your `update` method, assume that it takes two parameters in addition to `self`: a hypothesis and a piece of data, in that order. The hypothesis comes from the distribution, but the data may not.

Notes:

- `mean` and `quantile` only make sense when the hypotheses are numerical, but we can define them anyway.
- In the template, there is a placeholder method `likelihood` which is not implemented. This is because likelihood is specific to the problem. The placeholder is there simply so you can see the signature of the method, because `update` needs to call `likelihood`. `likelihood` will be defined in subclasses (below) after which `update` may work correctly.

1.3 Using our class to perform inference

In order to use the `InferenceSuite` class, it needs a `likelihood` method. But likelihoods are specific to the problem, which is why we did not implement this method in `InferenceSuite`. So, for each of the following, define a subclass of the `InferenceSuite` class with an associated `likelihood` method with the signature `likelihood(self, data, hypothesis)`

1.3.1 Problem 0: a worked example

On an in-class worksheet, we solved the following problem by hand:

> Three candidates run for election as a mayor in a city. A polling firm estimates that candidates A, B, and C have probabilities 0.25, 0.35, and 0.40 to be elected, respectively. You also estimate that these candidates each have a probability 0.6, 0.9, and 0.8 of building a certain bridge if elected.
>
> You live in a different city and don't hear who won the election, but a year later, you see in the news that the bridge has been built. What are the new probabilities that each of the candidates A, B, and C were elected?

The template `inference.py` contains a working subclass `Mayor` which encodes this problem. Note that the `__init__` method is modified to set the prior distribution to

```
{'A':0.25, 'B':0.35, 'C':0.40}
```

and the `likelihood` method takes a Boolean parameter as its `data`, with `True` representing the evidence "the bridge was built" and `False` representing "the bridge was not built". The return value of the `likelihood` method is one of the three values 0.6, 0.9, 0.8, or one of their complements.

Once you have defined the `update` method for the parent class `InferenceSuite`, the `Mayor` class will begin working and you can verify that calling `update(data)` with `data` either `True` or `False` will update the internal dictionary `self.d` to the correct probabilities:

```
update(True)  --> {'A':0.191, 'B':0.401, 'C':0.408}
update(False) --> {'A':0.465, 'B':0.163, 'C':0.372}
```

(float values rounded to three places)

1.3.2 Problem 1: The diagnostic test (10 points)

For the first problem, implement a version of a disease diagnostic test, similar to the cancer test discussed in class (on the topic of "detectors"). Your subclass should be called `Diagnostic`. You may leave the `__init__` method unmodified, but expect that the distribution of hypotheses will be initialized to a dictionary of the form

```
{'sick': p, 'healthy': 1 - p}
```

where `p` is the frequency of the disease in the general population.

Define a `likelihood` method whose signature is `likelihood(self, data, hypothesis)`; `data` is the result of a test for the disease, either `'+'` or `'-'`. For the likelihood model, you may assume:

- if the patient has the disease, the test result will be `'+'` with probability 0.9 and `'-'` with probability 0.1 (i.e., 10% false negative rate, or 90% sensitivity)
- if the patient does not have the disease, the test result will be `'+'` with probability 0.05 and `'-'` with probability 0.95 (i.e., 5% false positive rate, or 95% specificity)

1.3.3 Problem 2: the augmented cookie problem (10 points)

We have two bowls of cookies, each of which contains a known mixture of chocolate, vanilla, and strawberry cookies. We wish to estimate which bowl we are drawing from. Like in Written Homework 3, we will assume that the bowls are large enough that drawing cookies does not measurably alter the proportion.

In a twist on the base `InferenceSuite` class, the `Cookie` class will have additional instance variables which store the distribution of chocolate, vanilla, and strawberry cookies in each bowl. These variables will be initialized by a modified `__init__` method.

Your subclass must:

- be called `Cookie`
- replace the `__init__` method with version that takes two parameters `bowl1, bowl2` which represent the cookie flavor distributions in each bowl, in the form of a list or tuple. In the `__init__` method, initialize `self.d` to
  `{1: 0.5, 2: 0.5}`
  representing a uniform prior distribution on the two bowls, and then assign the parameters `bowl1, bowl2` to their own instance variables. You may assume that the proportions in each argument sum to 1 and that they are in the order `[chocolate, vanilla, strawberry]`.
- define a `likelihood` method which takes two parameters: a hypothesis (i.e. one of the outcomes from the dictionary, representing Bowl 1 or Bowl 2), and a piece of data which is `'c'` (chocolate), `'v'` (vanilla), or `'s'` (strawberry). It should return the probability of drawing the given type of cookie, assuming that we are drawing from the given bowl.

1.3.4 Problem 3: the locomotive problem (20 points: 10 class implementation, 10 experimentation)

In this problem, we do some parameter estimation.

Recall the locomotive problem from lecture: we are attempting to estimate the total number of trains, N, produced by a given railroad company, based on observing the existence of one or more numbered trains. We made the assumption that we are equally likely to observe any trains that actually exist, which implies the likelihood model:

$$P(\text{observe train } k | \text{there are } N \text{ total trains}) = \begin{cases} \frac{1}{N} & k \leq N \\ 0 & k > N \end{cases}$$

Use this to define the method `likelihood(self, data, hypothesis)`. Here, `data` is the number of the observed train, k, and the `hypothesis` is the hypothesized number of total trains, N.

Define a subclass called `Locomotive` that implements this likelihood model. Leave the `__init__` method un-initialized.

Then, once your `Locomotive` class is working, define a `main` function that does the following:

- initializes at least three instances of your `Locomotive` class with different prior distributions. You may use any prior distributions, including ones we have discussed in class.
- updates the distribution with the following observations: `187, 169, 299`
- computes and prints the following estimates of the true number N of trains for each prior: maximum a posteriori estimate, posterior mean, and 90% credible interval.

In your script, write a brief comment comparing the estimates you got for the different priors. Which combination of estimate and prior got closest to the true value?[1] Which priors gave a 90% credible interval that contains the true value?

1.4 Testing

A test script will be provided that tests your `InferenceSuite`, `Diagnostic`, `Cookie`, and `Locomotive` classes. The final "experimentation" component of the locomotive problem will be human-evaluated, since it is dependent on your choices of prior.

[1]The value of N used to generate the three observations was 387.