Chris Piech
CS 109

Section 6

1. **Binary Tree**: Consider the following function for constructing binary trees:

```
def random_binary_tree(p):
    """
␣␣␣␣Returns␣a␣dictionary␣representing␣a␣random␣binary␣tree␣structure.
␣␣␣␣The␣dictionary␣can␣have␣two␣keys,␣"left"␣and␣"right".
␣␣␣␣"""
    if random_bernoulli(p):   # returns true with probability p
        new_node = {}  # initialize one new node
        new_node["left"] = random_binary_tree(p)
        new_node["right"] = random_binary_tree(p)
        return new_node
    else:
        return None
```

The `if` branch is taken with probability p (and the `else` branch with probability $1 - p$). A tree with no nodes is represented by `None`; so a tree node with no left child has `None` for the `left` field (and the same for the right child).

Let X be the number of nodes in a tree returned by `random_binary_tree(p)`. You can assume $0 < p < 0.5$. What is $E[X]$, in terms of p?

The number of nodes in the tree depends on whether or not the if statement is true or false. It is true with probability p and false with probability 1 - p. This suggests that in order to find $E[X]$, we need to define a background random variable, Y, corresponding to the result of `random_bernoulli(p)`, where $P(Y = 1) = p$ and $P(Y = 0) = 1 - p$. Then, we can apply the Law of Total Expectation:

$$E[X] = p \cdot E[X \mid \text{if}] + (1 - p)E[X \mid \text{else}]$$

$$E[X] = p \cdot E[X \mid Y = 1] + (1 - p)E[X \mid Y = 0]$$

$E[X \mid Y = 0] = 0$ because if the `else` statement is executed, we return a tree with no nodes.

Let X_1 and X_2 be number of nodes returned by the left and right calls to `random_binary_tree`. We can write $E[X \mid Y = 1]$ as $E[1 + X_1 + X_2]$ because that represents the total number of nodes that are added to the tree if the `if` statement is true. Then, because the recursive call is identical to the original function call, we know that $E[X_1] = E[X_2] = E[X]$, so

$$E[X \mid Y = 1] = E[1 + X_1 + X_2] = 1 + E[X] + E[X] = 1 + 2E[X]$$

Putting this all together:

$$\begin{aligned} E[X] &= p \cdot E[X \mid Y = 1] + (1-p)E[X \mid Y = 0] \\ &= p \cdot E[1 + X_1 + X_2] + (1-p) \cdot 0 \\ &= p \cdot (1 + 2E[X]) \\ (1-2p)E[X] &= p \qquad \text{(getting } E[X] \text{ alone on the LHS)} \\ E[X] &= \frac{p}{1-2p} \end{aligned}$$

Extra Challenge Q: Why did we need to assume that p is less than 0.5?

2. **Entropy & Name2Age**

   See the Colab notebook at: `https://web.stanford.edu/class/cs109/section/6/`

   Solutions are available through a link on the course website page.

3. **Choosing a Diagnostic Test with Information Theory**

   A doctor is deciding which diagnostic test to administer to a patient. There are nine mutually exclusive possibilities: diseases A, B, C, D, E, F, G, H, or having *no disease* (labeled None).

   You are given:

   - `prior`: A prior distribution $P(x)$ over all diseases $x \in A, B, C, D, E, F, G, H$, None stored in a dictionary called prior.
   - A function `prob_pos_given_disease(test, x)` that returns the probability that a given test yields a positive result if the patient truly has disease x. (For the "None" case, this value represents the false-positive rate.)
   - `tests`: A list of available tests, stored as `tests`.

   When a test is run, it will return either a + or − result. However, the probability that a test is positive can vary depending on which disease the patient actually has. For example, a test designed to detect disease A may also occasionally return a positive result if the patient has disease B (a cross-reactivity), even though the true disease is not A. So while we only run one test at a time and get one result, that result provides evidence that updates our belief about *all* diseases.

   The goal is to determine which test to run by choosing the one that is expected to reduce our uncertainty about the patients condition the most. To do this, write code to compute the expected uncertainty for each test.

Let X be the (mutually exclusive) disease label which can be one of the values in D where $D = \{A, B, C, D, E, F, G, H, \text{None}\}$ and prior $P(x) = P(X = x)$. For test j, define the likelihood

$$\ell_j(x) \equiv P(+ \mid \text{test}_j,\ X = x) = \texttt{prob\_pos\_given\_disease}(\text{test}_j, x).$$

For each test, we need to know the probability that it returns positive and negative results.

$$P(+ \mid \text{test}_j) = \sum_{x \in D} P(x)\, \ell_j(x), \qquad P(- \mid \text{test}_j) = 1 - P(+ \mid \text{test}_j).$$

We have access to the probability that a test returns positive given the patient truly has that disease, but we want it in the other direction: probability the patient truly has that disease given the test results. So we use Bayes' rule.

$$P(X = x \mid +, \text{test}_j) = \frac{P(x)\, \ell_j(x)}{\sum_{k \in D} P(k)\, \ell_j(k)}, \qquad P(X = x \mid -, \text{test}_j) = \frac{P(x)\, [1 - \ell_j(x)]}{\sum_{k \in D} P(k)\, [1 - \ell_j(k)]}.$$

Posterior entropies and expected entropy.

$$\mathrm{H}\big(X \mid +, \text{test}_j\big) = -\sum_{x \in D} P(X = x \mid +, \text{test}_j)\, \log_2 P(X = x \mid +, \text{test}_j),$$

$$\mathrm{H}\big(X \mid -, \text{test}_j\big) = -\sum_{x \in D} P(X = x \mid -, \text{test}_j)\, \log_2 P(X = x \mid -, \text{test}_j),$$

$$\mathbb{E}\big[\mathrm{H}(X) \mid \text{test}_j\big] = P(+ \mid \text{test}_j)\, \mathrm{H}\big(X \mid +, \text{test}_j\big) + P(- \mid \text{test}_j)\, \mathrm{H}\big(X \mid -, \text{test}_j\big).$$

Doing this math for all of the diseases for all of tests by hand would take forever. Good thing we are computer scientists!! To the code:

```
import math

# note, the functions on the next line would depend on actual datasets!!
    This problem is just to practice writing the code. You will practice
    with real data on your next pset!!

from utils import load_from_data, prob_pos_given_disease

prior, tests = load_from_data()

def normalize_pmf(pmf):
    total = sum(pmf.values())
    for x in pmf:
        pmf[x] /= total
    return pmf

def compute_uncertainty(disease_pmf):
    H = 0.0
    for x in disease_pmf:
        p = disease_pmf[x]
        if p > 0:
            H += -p * math.log2(p)
```

```
        return H

    def compute_prob_pos(disease_pmf, test):
        # P(+ | test) = sum_x P(x) * P(+ | test, x)
        p_pos = 0.0
        for x in disease_pmf:
            p_pos += disease_pmf[x] * prob_pos_given_disease(test, x)
        return p_pos

    def compute_posterior(disease_pmf, test, outcome_is_positive):
        posterior_unnorm = {}
        for x in disease_pmf:
            like = prob_pos_given_disease(test, x)
            like = like if outcome_is_positive else (1 - like)
            posterior_unnorm[x] = disease_pmf[x] * like
        return normalize_pmf(posterior_unnorm)

    def compute_expected_uncertainty(disease_pmf, test):
        # E[H | test] = P(+) H(P(.|+,test)) + P(-) H(P(.|-,test))
        p_pos = compute_prob_pos(disease_pmf, test)
        p_neg = 1 - p_pos

        pmf_pos = compute_posterior(disease_pmf, test, True)  # outcome: +
        pmf_neg = compute_posterior(disease_pmf, test, False) # outcome: -

        H_pos = compute_uncertainty(pmf_pos) if p_pos > 0 else 0.0
        H_neg = compute_uncertainty(pmf_neg) if p_neg > 0 else 0.0

        return p_pos * H_pos + p_neg * H_neg

    def chose_best_test(disease_pmf, tests):
        best_test = None
        best_uncertainty = None
        for test in tests:
            expected_uncertainty = compute_expected_uncertainty(disease_pmf,
                test)
            if best_test is None or expected_uncertainty < best_uncertainty:
                best_test = test
                best_uncertainty = expected_uncertainty
        return best_test, best_uncertainty
```

4. **Variance of Height among Island Corgis (Optional)**

   This problem is great if you want more practice with bootstrapping but we aren't going to do it in section this week.

   A colleague has collected samples of heights of corgis that live on two different islands, A and

B. The colleague collects 50 samples from each island.

The sample mean is the same for both groups: 10 inches. However, island B has a **sample variance** that is 3.1 in^2 **greater** than island A. The colleague wants to make the claim that island B corgis have a significantly higher spread of heights than island A corgis. You are skeptical. It's possible that heights are identically distributed across both islands, and the observed difference in variance is just a result of chance and small sample size, i.e. the **null hypothesis**.

Write code that uses **bootstrapping** to calculate the probability of the null hypothesis. Here is the data. Each number is the height, in inches, of an independently sampled corgi:

Island A Corgi Heights ($S^2 = 6$): [13, 12, 7, 16, 9, 11, 7, 10, 9, 8, 9, 7, 16, 7, 9, 8, 13, 10, 11, 9, 13, 13, 10, 10, 9, 7, 7, 6, 7, 8, 12, 13, 9, 6, 9, 11, 10, 8, 12, 10, 9, 10, 8, 14, 13, 13, 10, 11, 12, 9]

Island B Corgi Heights ($S^2 = 9.1$): [8, 8, 16, 16, 9, 13, 14, 13, 10, 12, 10, 7, 14, 8, 13, 14, 7, 13, 7, 9, 4, 11, 7, 12, 8, 9, 12, 8, 11, 10, 12, 6, 10, 15, 11, 12, 3, 8, 11, 10, 10, 8, 12, 9, 11, 6, 7, 10, 9, 7]