1 Sampling

Lab Objective: *Sampling is an important and fundamental tool in statisical modeling. In this lab we will learn to use PyMC3 for Bayesian modeling and statical sampling.*

Sampling

When seeking to understand a group or a phenomenon, we, as data scientists, will often look to find some sort of sample. A good sample can tell us a lot, and while we will not examine what makes a good sample in this lab, we will examine how much a good sample can tell us. One goal of Bayesian statistics is to be able to quantify, with degrees of certainty, how much we can learn from a sample, given what we already know about its source. This quantification of certainty allows us to extract more information and nuance from a sample than would otherwise be possible, and in turn allow us to better predict and describe events.

Parameter Estimation

Maximum Likelihood Estimation

Maximum Likelihood Estimation is a frequentist approach to parameter estimation. We will first examine the derivation of the maximum likelihood estimate (MLE) from the frequentist point of view, then examine how it is a special case of the Bayesian method of finding the *maximum a posteriori estimate* (MAP).

Likelihood

Finding the maximum likelihood involves, unsurprisingly, maximizing the likelihood function, which is defined as follows

$$\mathcal{L}(\theta) = \mathcal{L}(\theta|\mathbf{x}) = f(\mathbf{x}|\theta),$$

where $\mathbf{x}$ is a sample, θ is the parameter we are estimating and f is the pdf. Determining the MLE $\hat{\theta}$ is as simple as finding the argmax of $\mathcal{L}$

$$\hat{\theta} = \text{argmax}_{\theta\in\Theta}\mathcal{L}(\theta|\mathbf{x}).$$

Maximum A Posteriori Estimate

If we examine closely, we can see a similarity between the likelihood function and Bayes' rule. Bayes' rule gives the following relation

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

To apply this rule to the problem of parameter estimation we get the following relation

$$f(\theta|\mathbf{x}) = \frac{f(\mathbf{x}|\theta)g(\theta)}{\int_\Theta f(\mathbf{x}|\vartheta)g(\vartheta)d\vartheta}. \tag{1.1}$$

We call $f(\theta|\mathbf{x})$ the posterior distribution and $g(\theta)$ the prior distribution. The MAP estimate is then the argmax of the posterior

$$\theta_{MAP} = \text{argmax}_{\theta\in\Theta} f(\theta|\mathbf{x}). \tag{1.2}$$

When finding the MAP estimate, the exact posterior is often left uncalculated because it is difficult to compute. Instead we approximate it by finding its value at grid points of θ. Similarly, because of the complexity of the denominator, we often don't find it until the end of the process. After we calculate $f(\mathbf{x}|\theta)g(\theta)$ for each relevant θ we can then approximate the integral in the denominator with a finite sum. First we find $f(\mathbf{x}|\theta_i)g(\theta_i)$ for a grid of θ values. Then, we can approximate the integral in denominator with the following sum $\sum_i f(\mathbf{x}|\theta_i)g(\theta_i)$.

Here we can see that the likelihood function is similar to Bayes' rule as long as we take $g(\theta)$ to be a constant, i.e. $\theta \sim \mathcal{U}(a, b)$. This means that the MAP estimate is the MLE if we assume a uniform prior distribution.

Problem 1. Write a function called `bernoulli_sampling()` that takes the following parameters: `p` a float that is the "fairness" of a coin and `n` the size of the sample to be generated. In this function simulate `n` tosses of a coin which gives heads with probability `p`. Then use that sample to calculate the posterior distribution on p given a uniform prior using Equation 1.2.

For `p`=.2 and `n`=100 plot the posterior distribution and return the MAP estimate of p, which is also the MLE in this case.

Hint: In this case f is the Binomial pmf $f(x) = p^{nx}(1-p)^{n(1-x)}$. You do not need to calculate the integral in the denominator exactly; since you are using a finite approximation of the distributions, you may use a finite approximation of the integral. You may simulate the tosses of a coin by using `np.random.binomial()` or `scipy.stats.binom()`.

Non-Uniform Priors

While we are able to get good estimates, we leave a lot of the power of Bayesian statistics on the table when we only use a uniform prior. While the uniform prior is free from any preconceptions or biases, it also imparts the least amount of information. Using a non-uniform prior allows us to actually incorporate prior knowledge or assumptions into our model. If we have good reason to believe something about a parameter we are exploring before we even draw a sample, we can learn a lot more by accounting for those beliefs.

Problem 2. Suppose you choose a coin from a bag that produces coins of many weights. However, the bag seems to be more likely to produce coins that are strongly biased in favor of heads. You're unsure of which kind of coin you've drawn so in order to find out you perform 20 flips.

Write a function called `non_uniform_prior()` that takes the following parameters: `p` a float that is the "fairness" of a coin, `n` the size of the sample to be generated, and `prior` a SciPy distribution object which will act as the prior on p.

Similar to Problem 1, simulate `n` flips and calculate and plot the posterior distribution. Return the MAP estimate.

Examine the difference in confidence we can have in estimating the bias of the coin if the coin we draw gives heads 90% of the time as opposed to 40% of the time.

Because we think that coins biased in favor of heads are likely, we can choose a prior distribution that matches that assumption. In this case we will choose Beta$(5, 1.5)$ as the prior distribution because it gives much more weight to parameters larger than .5. This is most easily achieved with `scipy.stats.beta(5,1.5)` and using the `pdf()` method to calculate $g(\theta)$

Sampling from a Markov Chain

A Markov chain is a way to model sequences of states or events. Markov chains make a few assumptions, one of those being that the probability of each state occurring is dependent only on the previous state. The relationship between the states are described by what is called a transition matrix.

Markov chains and sampling are like peanut butter and jelly: neither one really lives up to their full potential without the other. Given the transition matrix of a Markov chain, we can use sampling to better understand what that chain looks and acts like or to get a well-informed idea of what the future may hold.

Sampling from a simple (row stochastic) transition matrix like the one below is as simple as picking a starting state s_0, and then using the corresponding row to sample randomly using the probabilities in the row.

	a	b	c
a	0.7	0.1	0.2
b	0.5	0.4	0.1
c	0.1	0.8	0.1

For example, using the above transition matrix, let $s_0 = a$. Then I will randomly sample from the array $[a, b, c]$ using the respective probabilities $[0.7, 0.1, 0.2]$. If the sample gives me c, then. $s_1 = c$ and we can continue the process to find $s_2, ..., s_n$.

Problem 3. Given the transition matrix below and assuming the 0th day is sunny, sample from the markov chain to give a possible forecast of the 10 following days. Return a list of strings.

	sun	rain	wind
sun	0.6	0.1	0.3
rain	0.2	0.6	0.2
wind	0.3	0.4	0.3

Hint: `np.random.choice()` may be helpful here.

PyMC3

Python has many powerful sampling tools. Among these is PyMC3, an efficient implementation of a method known as Monte Carlo Markov Chain (MCMC) Sampling. This is a useful technique as it constructs a Markov Chain whose steady state is a probability distribution that is difficult to sample from directly. Unlike our simple Markov Chain from the last problem, certain Markov Chains are abstract. PyMC3 gives us a way to work with these more complex scenarios.

Single Variable PyMC3

Consider the following: owners of a restaurant are trying to decide if they should keep selling nachos. They gather the data for several months about how many people order nachos each day. One of the owners happened to take a class in Bayesian statistics in college, so she decides test her knowledge. She assumes the data are distributed as Poisson(λ) for some unknown value of λ where λ has a prior of Gamma(2,2). She sets up a PyMC3 Model for the situation as follows:

```
import pymc3 as pm
import numpy as np
import arviz as az #visualization package

model = pm.Model()
with model:
    lam = pm.Gamma('lambda', alpha=2, beta=2)
    y = pm.Poisson('y', mu=lam, observed=nacho_data)
    trace = pm.sample(n) #n is the desired number of samples

    az.plot_trace(trace)
    lam = trace['lambda']
    mean = lam.mean()
```

What this has done is created a model for the Poisson Distribution as well as λ itself. It then samples from the posterior to give us the expected value of λ (mean). It also gives a sample trace of length n from posterior, and the trace plot.

We will now reconsider the initial problem of the coin flip.

Problem 4. Create a function the accepts the coin flip data in array form and an integer n for desired number of samples. Given data that flips a coin 100 times, assume the data are distributed as Bernoulli(p) for some unknown value of p, where p has a prior of Beta(1,1). Set up a PyMC3 model for this situation and sample from the posterior n times. Print a trace plot. Return the mean for the posterior.

Run the function with data generated by the following code

```
from scipy.stats import bernoulli
data = bernoulli.rvs(0.2, size=30)
```

Multivariate PyMC3

Unlike the Poisson and Bernoulli distributions, many other distributions including the Normal, Beta, Gamma, and Binomial distributions have two or more parameters. These problems are where PyMC3 becomes very useful.

Problem 5. Create a function the accepts the height data in array form and an integer n for desired number of samples. Given a dataset of the measured heights of 100 men, assume the data are distributed as Normal(mu,1/tau) where mu has a prior of Normal(m,s), and tau has a prior of Gamma(alpha,beta). Set up a PyMC3 model for this situation and sample from the posterior n times. Print a trace plot for mu and tau. Return the mean for the posterior of mu.

Run the function with data generated by the following code

```
heights = np.random.normal(180,10,100)
```