

### Problem 6.

first derivative:  $\frac{d}{\theta} [\theta^t (1-\theta)^h] = t \theta^{t-1} (1-\theta)^h - h \theta^t (1-\theta)^{h-1}$

second derivative:  $\frac{d}{\theta} [t \theta^{t-1} (1-\theta)^h - h \theta^t (1-\theta)^{h-1}] = t(t-1) \theta^{t-2} (1-\theta)^h - ht \theta^{t-1} (1-\theta)^{h-1}$   
 $-ht \theta^{t-1} (1-\theta)^{h-1} + h(h-1) \theta^t (1-\theta)^{h-2}$   
 $= t(t-1) \theta^{t-2} (1-\theta)^h - 2ht \theta^{t-1} (1-\theta)^{h-1} + h(h-1) \theta^t (1-\theta)^{h-2}$

first derivative of  $\log$ :  $\frac{d}{\theta} \log [\theta^t (1-\theta)^h] = \frac{d}{\theta} [t \log \theta + h \log (1-\theta)] = \frac{t}{\theta} - \frac{h}{1-\theta}$

second derivative of  $\log$ :  $\frac{d}{\theta} \left( \frac{t}{\theta} - \frac{h}{1-\theta} \right) = -\frac{t}{\theta^2} + \frac{h}{(1-\theta)^2}$

### Problem 7.

every local maximum of  $\log f(\theta)$ :  $\frac{d}{\theta} \log f(\theta) = 0 \Rightarrow \frac{f'(\theta^*)}{f(\theta^*)} = 0$ , because  $f(\theta) > 0$   
 so  $f(\theta^*) > 0 \neq 0 \Rightarrow f'(\theta^*) = 0$ , first derivative of  $\log f(\theta)$  is 0.

also means first derivative of  $f(\theta^*)$  is 0  $\Rightarrow$  is also a local maximum of  $f(\theta)$

conclusion: The logarithm simplifies the computation and preserves the monotonicity and local extrema of original function.

### Problem 8

prior distribution:  $p(\theta) = \text{Beta}(\theta | a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \theta^{a-1} (1-\theta)^{b-1}$

posterior distribution is also Beta distribution:  $p(\theta | D) = \text{Beta}(\theta | a+m, b+l)$

posterior mean value  $E[\theta | D] = \frac{a+m}{a+b+m+l}$

MLE:  $\theta_{\text{MLE}} = \frac{m}{m+l}$ , prior mean value:  $E[\theta] = \frac{a}{a+b}$

from question:  $E[\theta | D] = \lambda E[\theta] + (1-\lambda) \theta_{\text{MLE}}$   
 $= \lambda \frac{a}{a+b} + (1-\lambda) \frac{m}{m+l}, 0 \leq \lambda \leq 1$

because  $E[\theta | D] = \frac{a}{a+b+m+l} + \frac{m}{a+b+m+l}$

$\Rightarrow \lambda = \frac{a}{a+b+m+l}, (1-\lambda) = \frac{m}{a+b+m+l} \Rightarrow 0 \leq \lambda \leq 1, 0 \leq 1-\lambda \leq 1$

$\Rightarrow$  posterior mean is a compromise between the prior distribution and the maximum likelihood solution.

### Problem 9.

posterior distribution:  $P(\lambda | x, a, b) = P(\lambda | a, b) \cdot P(x | \lambda) / P(x)$   
 $\propto P(\lambda | a, b) \cdot P(x | \lambda) \propto \left( \frac{1}{\Gamma(a)} \lambda^{a-1} e^{-b\lambda} \right) \cdot \left( \frac{\lambda^x e^{-\lambda}}{x!} \right) \xrightarrow{\text{eliminate const.}}$

$\propto \lambda^{x+a-1} e^{-(b+1)\lambda} \Rightarrow$  also Gamma distribution:  $\text{Gamma}(\lambda | x+a, b+1)$ .

$\Rightarrow \lambda_{\text{MAP}} = \frac{a-1}{b} = \frac{x+a-1}{b+1}$

# Programming Task: Probabilistic Inference

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

from scipy.special import loggamma
%matplotlib inline
```

## Your task

This notebook contains code implementing the methods discussed in Lecture 3: Probabilistic Inference. Some functions in this notebook are incomplete. Your task is to fill in the missing code and run the entire notebook.

In the beginning of every function there is docstring which specifies the input and expected output. Write your code in a way that adheres to it. You may only use plain python and anything that we imported for you above such as `numpy` functions (i.e. no scikit-learn classifiers).

## Simulating data

The following function simulates flipping a biased coin.

```
In [2]: # This function is given, nothing to do here.
def simulate_data(num_samples, tails_proba):
    """Simulate a sequence of i.i.d. coin flips.

    Tails are denoted as 1 and heads are denoted as 0.

    Parameters
    -----
    num_samples : int
        Number of samples to generate.
    tails_proba : float in range (0, 1)
        Probability of observing tails.

    Returns
    -----
    samples : array, shape (num_samples)
        Outcomes of simulated coin flips. Tails is 1 and heads is 0.
    """
    return np.random.choice([0, 1], size=(num_samples), p=[1 - tails_proba, tail
```

```
In [3]: np.random.seed(123) # for reproducibility
num_samples = 20
tails_proba = 0.7
samples = simulate_data(num_samples, tails_proba)
print(samples)
```

```
[1 0 0 1 1 1 1 1 1 1 0 1 1 0 0 1 1]
```

## Important: Numerical stability

When dealing with probabilities, we often encounter extremely small numbers. Because of limited floating point precision, directly manipulating such small numbers can lead to serious numerical issues, such as overflows and underflows. Therefore, we usually work in the **log-space**.

For example, if we want to multiply two tiny numbers  $a$  and  $b$ , we should compute  $\exp(\log(a) + \log(b))$  instead of naively multiplying  $a \cdot b$ .

For this reason, we usually compute **log-probabilities** instead of **probabilities**. Virtually all machine learning libraries are dealing with log-probabilities instead of probabilities (e.g. [Tensorflow-probability](https://www.tensorflow.org/probability) (<https://www.tensorflow.org/probability>)) or [Pyro](https://pyro.ai) (<https://pyro.ai>)).

## Task 1: Compute $\log p(\mathcal{D} \mid \theta)$ for different values of $\theta$

```
In [4]: def compute_log_likelihood(theta, samples):
    """Compute log p(D | theta) for the given values of theta.

    Parameters
    -----
    theta : array, shape (num_points)
        Values of theta for which it's necessary to evaluate the log-likelihood.
    samples : array, shape (num_samples)
        Outcomes of simulated coin flips. Tails is 1 and heads is 0.

    Returns
    -----
    log_likelihood : array, shape (num_points)
        Values of log-likelihood for each value in theta.
    """
    ### YOUR CODE HERE ####
    num_tails = np.sum(samples)
    num_heads = len(samples) - num_tails

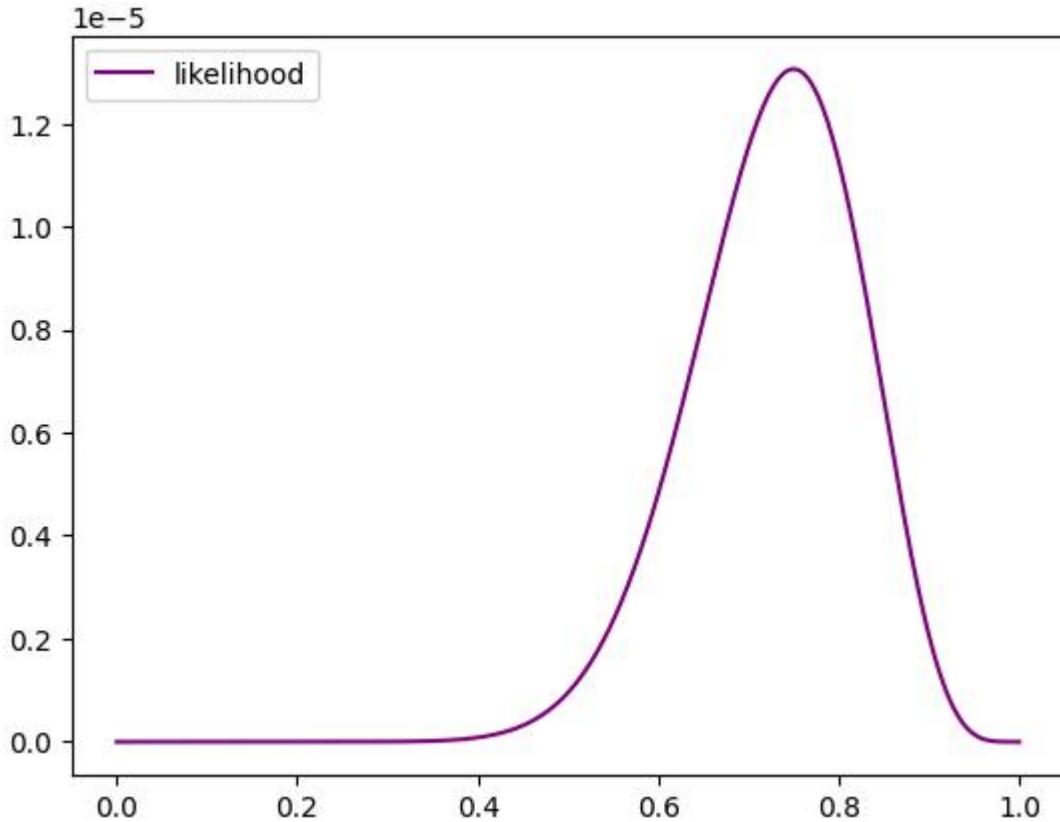
    log_likelihood = np.zeros(len(theta))

    for i, t in enumerate(theta):
        if t <= 0 or t >= 1:
            log_likelihood[i] = -np.inf
        else:
            log_likelihood[i] = num_tails * np.log(t) + num_heads * np.log(1 - t)

    return log_likelihood
```

```
In [5]: x = np.linspace(1e-5, 1-1e-5, 1000)
log_likelihood = compute_log_likelihood(x, samples)
likelihood = np.exp(log_likelihood)
plt.plot(x, likelihood, label='likelihood', c='purple')
plt.legend()
```

Out[5]: <matplotlib.legend.Legend at 0x14d5cc13730>



Note that the likelihood function doesn't define a probability distribution over  $\theta$  --- the integral  $\int_0^1 p(D | \theta) d\theta$  is not equal to one.

To show this, we approximate  $\int_0^1 p(D | \theta) d\theta$  numerically using [the rectangle rule](#) ([https://en.wikipedia.org/wiki/Riemann\\_sum](https://en.wikipedia.org/wiki/Riemann_sum)).

```
In [6]: # 1.0 is the length of the interval over which we are integrating p(D | theta)
int_likelihood = 1.0 * np.mean(likelihood)
print(f'Integral = {int_likelihood:.4f}' )
```

Integral = 3.068e-06

## Task 2: Compute $\log p(\theta | a, b)$ for different values of $\theta$

The function `loggamma` from the `scipy.special` package might be useful here. (It's already imported - see the first cell)

```
In [7]: def compute_log_prior(theta, a, b):
    """Compute log p(theta | a, b) for the given values of theta.

    Parameters
    -----
    theta : array, shape (num_points)
        Values of theta for which it's necessary to evaluate the log-prior.
    a, b: float
        Parameters of the prior Beta distribution.

    Returns
    -----
    log_prior : array, shape (num_points)
        Values of log-prior for each value in theta.

    """
    ### YOUR CODE HERE ###

    log_prior = np.zeros(len(theta))

    log_gamma_ab = loggamma(a + b)
    log_gamma_a = loggamma(a)
    log_gamma_b = loggamma(b)

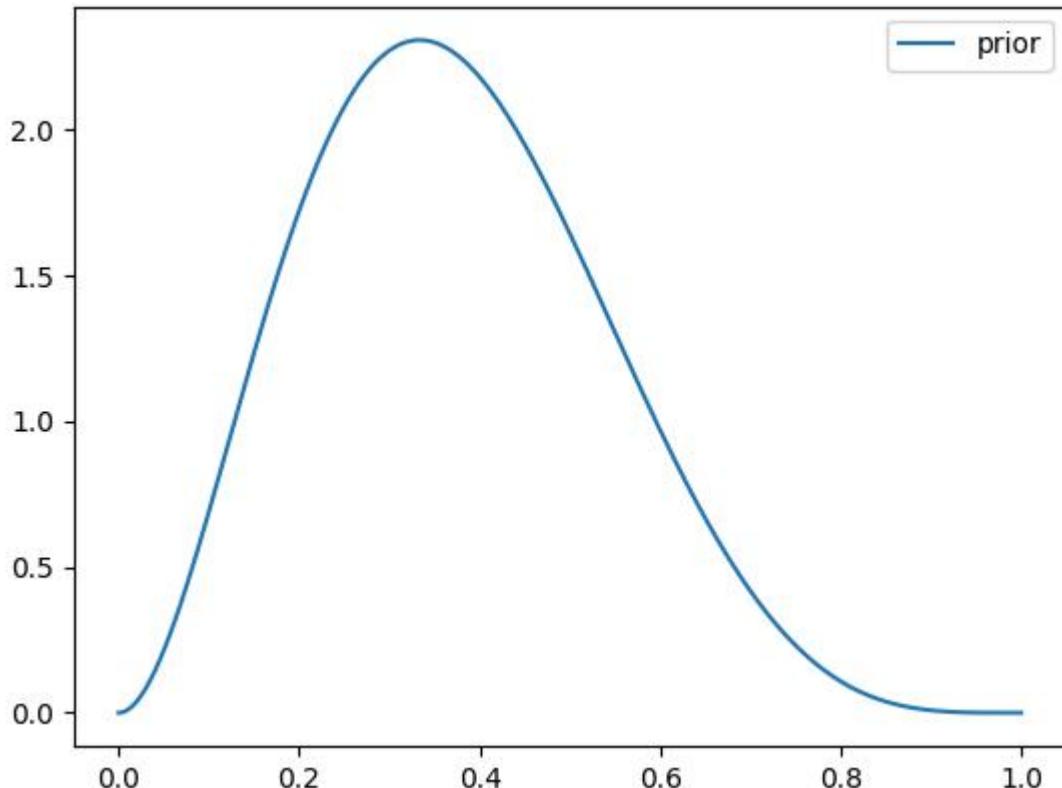
    for i, t in enumerate(theta):
        if t <= 0 or t >= 1:
            log_prior[i] = -np.inf
        else:
            log_prior[i] = (
                log_gamma_ab - log_gamma_a - log_gamma_b +
                (a - 1) * np.log(t) + (b - 1) * np.log(1 - t)
            )

    return log_prior
```

```
In [8]: x = np.linspace(1e-5, 1-1e-5, 1000)
a, b = 3, 5

# Plot the prior distribution
log_prior = compute_log_prior(x, a, b)
prior = np.exp(log_prior)
plt.plot(x, prior, label='prior')
plt.legend()
```

Out[8]: <matplotlib.legend.Legend at 0x14d5ee47ee0>



Unlike the likelihood, the prior defines a probability distribution over  $\theta$  and integrates to 1.

```
In [9]: int_prior = 1.0 * np.mean(prior)
print(f'Integral = {int_prior:.4f}')
```

Integral = 0.999

### Task 3: Compute $\log p(\theta | D, a, b)$ for different values of $\theta$

The function `loggamma` from the `scipy.special` package might be useful here.

```
In [10]: def compute_log_posterior(theta, samples, a, b):
    """Compute log p(theta | D, a, b) for the given values of theta.
```

Parameters

-----  
theta : array, shape (num\_points)  
Values of theta for which it's necessary to evaluate the log-prior.  
samples : array, shape (num\_samples)  
Outcomes of simulated coin flips. Tails is 1 and heads is 0.  
a, b: float  
Parameters of the prior Beta distribution.

Returns

-----  
log\_posterior : array, shape (num\_points)  
Values of log-posterior for each value in theta.

"""

### YOUR CODE HERE ###

```
log_likelihood_values = compute_log_likelihood(theta, samples)
```

```
log_prior_values = compute_log_prior(theta, a, b)
```

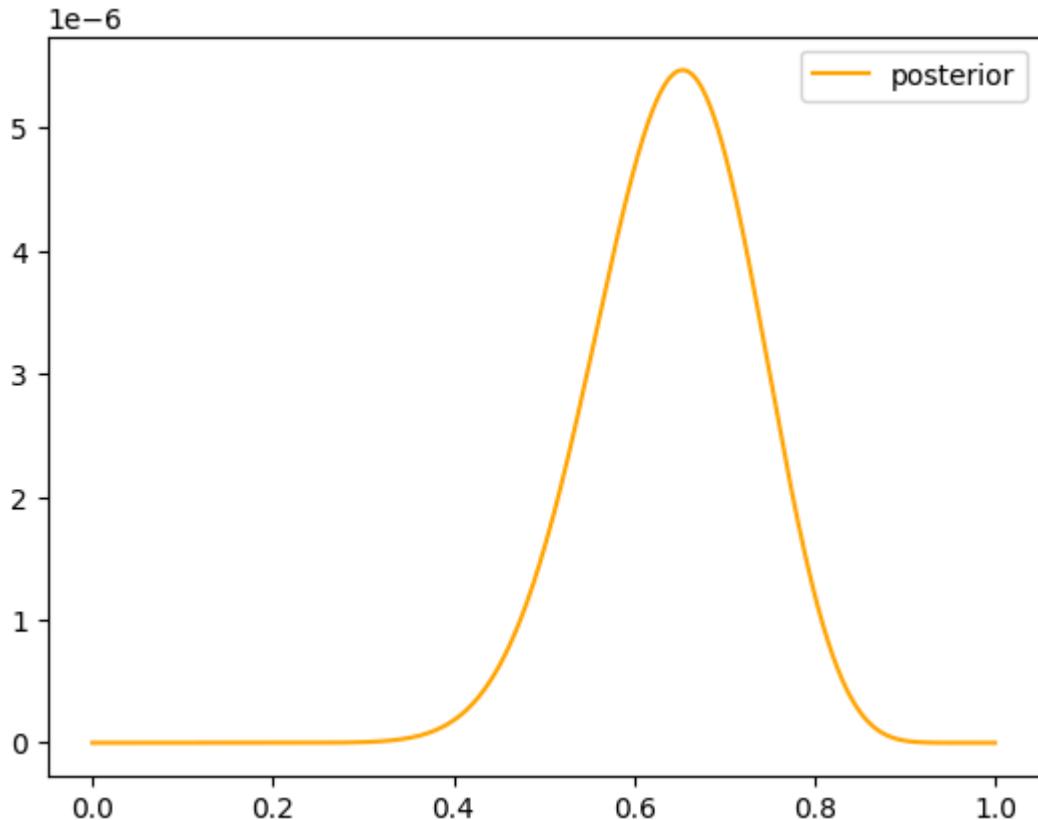
```
log_posterior_values = log_likelihood_values + log_prior_values
```

```
return log_posterior_values
```

```
In [11]: x = np.linspace(1e-5, 1-1e-5, 1000)

log_posterior = compute_log_posterior(x, samples, a, b)
posterior = np.exp(log_posterior)
plt.plot(x, posterior, label='posterior', c='orange')
plt.legend()
```

Out[11]: <matplotlib.legend.Legend at 0x14d5eec7e80>



Like the prior, the posterior defines a probability distribution over  $\theta$  and integrates to 1.

```
In [12]: int_posterior = 1.0 * np.mean(posterior)
print(f'Integral = {int_posterior:.4f}')
```

Integral = 1.243e-06

## Task 4: Compute $\theta_{MLE}$

```
In [13]: def compute_theta_mle(samples):
    """Compute theta_MLE for the given data.

    Parameters
    -----
    samples : array, shape (num_samples)
        Outcomes of simulated coin flips. Tails is 1 and heads is 0.

    Returns
    -----
    theta_mle : float
        Maximum likelihood estimate of theta.
    """
    ### YOUR CODE HERE ###

    theta_mle = np.sum(samples) / len(samples)

    return theta_mle
```

```
In [14]: theta_mle = compute_theta_mle(samples)
print(f' theta_mle = {theta_mle:.3f}' )
```

```
theta_mle = 0.750
```

## Task 5: Compute $\theta_{MAP}$

```
In [15]: def compute_theta_map(samples, a, b):
    """Compute theta_MAP for the given data.

    Parameters
    -----
    samples : array, shape (num_samples)
        Outcomes of simulated coin flips. Tails is 1 and heads is 0.
    a, b: float
        Parameters of the prior Beta distribution.

    Returns
    -----
    theta_mle : float
        Maximum a posteriori estimate of theta.
    """
    ### YOUR CODE HERE ###

    num_tails = np.sum(samples)

    num_samples = len(samples)

    theta_map = (a + num_tails - 1) / (a + b + num_samples - 2)

    if theta_map < 0:
        theta_map = 0
    elif theta_map > 1:
        theta_map = 1

    return theta_map
```

```
In [16]: theta_map = compute_theta_map(samples, a, b)
print(f'theta_map = {theta_map:.3f}')
```

```
theta_map = 0.654
```

## Putting everything together

Now you can play around with the values of `a`, `b`, `num_samples` and `tails_proba` to see how the results are changing.

```
In [17]: num_samples = 20
tails_proba = 0.7
samples = simulate_data(num_samples, tails_proba)
a, b = 3, 5
print(samples)
```

```
[1 1 1 1 1 1 0 0 1 0 1 1 1 1 1 1 1]
```

```
In [18]: plt.figure(figsize=[12, 8])
x = np.linspace(1e-5, 1-1e-5, 1000)

# Plot the prior distribution
log_prior = compute_log_prior(x, a, b)
prior = np.exp(log_prior)
plt.plot(x, prior, label='prior')

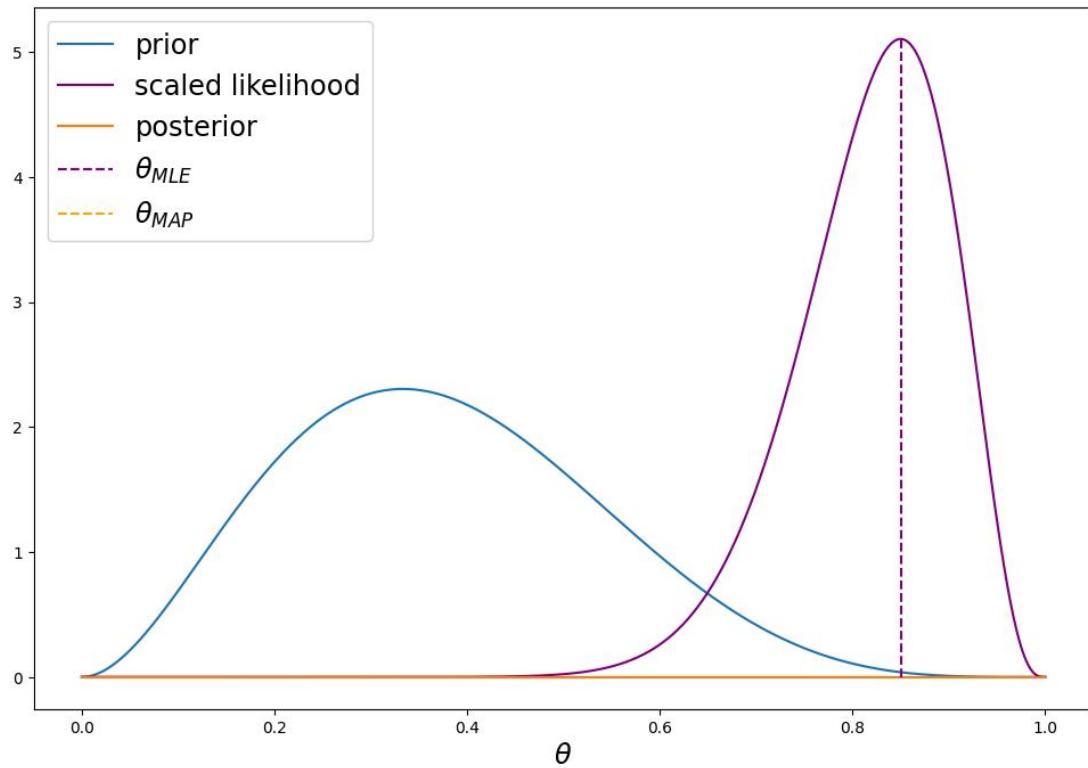
# Plot the likelihood
log_likelihood = compute_log_likelihood(x, samples)
likelihood = np.exp(log_likelihood)
int_likelihood = np.mean(likelihood)
# We rescale the likelihood - otherwise it would be impossible to see in the plot
rescaled_likelihood = likelihood / int_likelihood
plt.plot(x, rescaled_likelihood, label='scaled likelihood', color='purple')

# Plot the posterior distribution
log_posterior = compute_log_posterior(x, samples, a, b)
posterior = np.exp(log_posterior)
plt.plot(x, posterior, label='posterior')

# Visualize theta_mle
theta_mle = compute_theta_mle(samples)
ymax = np.exp(compute_log_likelihood(np.array([theta_mle]), samples)) / int_likelihood
plt.vlines(x=theta_mle, ymin=0.00, ymax=ymax, linestyle='dashed', color='purple',

# Visualize theta_map
theta_map = compute_theta_map(samples, a, b)
ymax = np.exp(compute_log_posterior(np.array([theta_map]), samples, a, b))
plt.vlines(x=theta_map, ymin=0.00, ymax=ymax, linestyle='dashed', color='orange',

plt.xlabel(r'$\theta$', fontsize='xx-large')
plt.legend(fontsize='xx-large')
plt.show()
```



In [ ]: