Machine Learning Exercise Sheet 1

Math Refresher

Group_369

Fan Xue – fan98.xue@tum.de

Xing Zhou – xing.zhou@tum.de

Jianzhe Liu – jianzhe.liu@tum.de

November 10, 2021

Problem 6

First let's compute the first and second derivative of $\theta^t(1-\theta)^h$, According to the derivative calculation rule:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}\theta}\theta^t(1-\theta)^h &= t\theta^{t-1}(1-\theta)^h + \theta^t(-1)h(1-\theta)^{h-1}\\ &= \theta^{t-1}(1-\theta)^{h-1} * ((1-\theta)t - \theta h)\end{aligned}$$

$$\begin{aligned}\frac{\mathrm{d}^2}{\mathrm{d}\theta^2}\theta^t(1-\theta)^h &= \frac{\mathrm{d}}{\mathrm{d}\theta}\left(t\theta^{t-1}(1-\theta)^h + \theta^t(-1)h(1-\theta)^{h-1}\right)\\ &= \theta^{t-2}(1-\theta)^{h-2} * ((1-\theta)(t-1) - \theta(h-1)) * ((1-\theta)t - \theta h) - \theta^{t-1}(1-\theta)^{h-1}(t+h)\end{aligned}$$

It's quite obvious that as the derivation goes on, the result becomes much more complex, which is really hard for us to read and analyse.

Then let's try computing the first and second derivative of $\log\theta^t(1-\theta)^h$:

$$f(\theta) = \log\theta^t(1-\theta)^h = t\log\theta + h\log(1-\theta)$$

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}\theta}f(\theta) &= \frac{t}{\theta} - \frac{h}{1-\theta}\\ \frac{\mathrm{d}^2}{\mathrm{d}\theta^2}f(\theta) &= \frac{\mathrm{d}}{\mathrm{d}\theta}\left(\frac{t}{\theta} - \frac{h}{1-\theta}\right)\\ &= -\frac{t}{\theta^2} - \frac{h}{(1-\theta)^2}\end{aligned}$$

This time the result is much more clear, and it is of course easier for us to do the further calculation or analysis.

Problem 7

To prove this, let's say we have an arbitrary local maximum θ_a in $\log f(\theta)$, obviously in any small range around θ_a, we have $\log f(\theta_a) \geq \log f(\theta)$.

Then we use function *exp* to transform $\log f(\theta)$ into $f(\theta)$.

Since function *exp* is monotone, we have:

$$f(\theta_a) = e^{\log f(\theta_a)} \geq f(\theta) = e^{\log f(\theta)}$$

Therefore θ_a is also a maximum of $f(\theta)$.

Conclusion: with the result of last problem, we know the using function *log* can help us simplify the calculation and thus the further analysis. And with the result of this problem we know that by using *log* transform, the feature of the original equations remain unchanged, which means we can safely apply this transform in further studies.

Problem 8

To proof this, we need to proof a Lemma first.
Lemma:
For $a, b, c > 0$, $0 < \lambda < 1$, if $c = \lambda a + (1 - \lambda)b$, then c lies between a and b.
Proof:
Assuming that $a < b$, we have

$$c - a = (\lambda - 1)a + (1 - \lambda)b = (1 - \lambda)(b - a) > 0c - b = \lambda a - \lambda b = \lambda(a - b) < 0$$

that indicates that c lies bewteen a and b.
Now we change bach to the problem.
According to the given information, we know that the **prior distribution** $p(\theta) = Beta(a, b)$, and the **likelihood** $p(D|\theta) = Binary(m, N, \theta)$. That is

$$p(\theta) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \theta^{a-1}(1-\theta)^{b-1}$$
$$p(D|\theta) = C_N^m \theta^m (1\ \theta)^{N-m}$$

We can know that the **posterior distribution**

$$p(\theta|D) \sim p(D|\theta)p(\theta) \sim \theta^{a+m-1}(1-\theta)^{b+N-m-1}$$

Obviously the **posterior distribution** is a Beta distribution, that is

$$p(\theta|D) \sim Beta(a + m, b + N - m)$$

Let $\log p(D|\theta) = 0$, we can find the maximum likelihood estimate

$$\theta_{MLE} = \frac{m}{N}$$

From the konwledge of Beta distribution, we know the posterior mean value of θ

$$\mathbb{E}(\theta|D) = \frac{a+m}{a+b+N}$$

and the prior mean of θ

$$\mathbb{E}(\theta) = \frac{a}{a+b}$$

We can rewrite the posterior mean of θ as following

$$\mathbb{E}(\theta|D) = \frac{a+m}{a+b+N} = \frac{N}{a+b+N} \cdot \frac{m}{N} + \frac{a+b}{a+b+N} \cdot \frac{a}{a+b} = \lambda\mathbb{E}(\theta) + (1-\lambda)\theta_{MLE}$$

From the lemma we can proof that $\mathbb{E}\theta$ lies between $\mathbb{E}(\theta)$ and θ_{MLE}.

Problem 9

The MAP estimation of the parameter λ is:

$$\begin{aligned}
\lambda_{MAP} &= \arg\max_{\lambda} \mathrm{p}(\lambda \mid x, a, b) \\
&= \arg\max_{\lambda} \log \mathrm{p}(\lambda \mid x, a, b) \\
&= \arg\max_{\lambda} \log(\mathrm{p}(x \mid \lambda)\, \mathrm{p}(\lambda \mid a, b)) \\
&= \arg\max_{\lambda} \log(\frac{b^a \lambda^{a-1} \exp(-b\lambda)}{\Gamma(a)} \frac{\lambda^x \exp(-\lambda)}{x!}) \\
&= \arg\max_{\lambda} (a-1+x)\log\lambda - (b+1)\lambda + \mathit{const}
\end{aligned}$$

In order to maximize the function, compute the derivative:

$$\frac{\partial}{\partial\lambda}((a-1+x)\log\lambda - (b+1)\lambda + \mathit{const}) = \frac{a-1+x}{\lambda} - b - 1 \overset{!}{=} 0$$

Then we get

$$\lambda = \frac{x+a-1}{b+1}$$

Hence $\lambda_{MAP} = \frac{x+a-1}{b+1}$.

exercise_03_prob_inference

November 9, 2021

1 Programming Task: Probabilistic Inference

[1]:

```
import numpy as np
import matplotlib.pyplot as plt

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

1.1 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 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).

1.2 Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is 1. Run all the cells of the notebook (`Kernel -> Restart & Run All`) 2. Export/download the notebook as PDF (`File -> Download as -> PDF via LaTeX (.pdf)`) 3. Concatenate your solutions for other tasks with the output of Step 2. On Linux you can simply use `pdfunite`, there are similar tools for other platforms too. You can only upload a single PDF file to Moodle.

Make sure you are using `nbconvert` **Version 5.5 or later** by running `jupyter nbconvert --version`. Older versions clip lines that exceed page width, which makes your code harder to grade.

1.3 Simulating data

The following function simulates flipping a biased coin.

[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,␣
 ↪tails_proba])
```

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

2 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 or Pyro).

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

[4]:

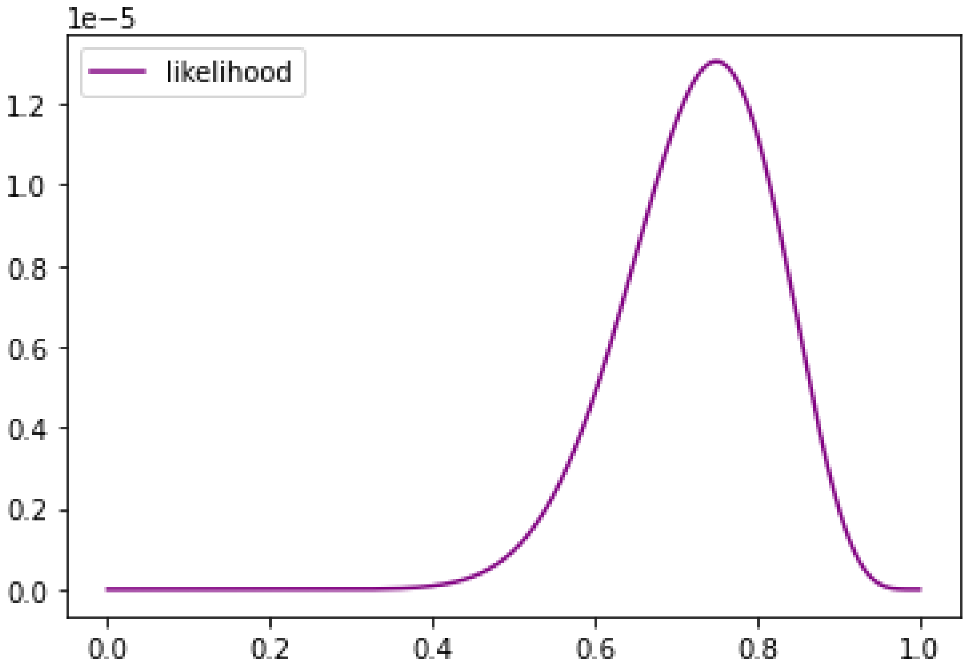
```python
def compute_log_likelihood(theta, samples):
    """Compute log p(D | theta) for the given values of theta.

    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_points = len(theta)
    num_samples = len(samples)

    num_tails, num_heads = 0, 0
    for sample in samples:
        if sample == 1:
            num_tails = num_tails + 1
        else:
            num_heads = num_heads + 1
    log_likelihood = num_tails * np.log(theta) + num_heads * np.log(1 - theta)
    return log_likelihood
```

[5]:

```python
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()
```

[5]:

```
<matplotlib.legend.Legend at 0x7f837ab630d0>
```



Note that the likelihood function doesn't define a probability distribution over θ — the integral $\int_0^1 p(\mathcal{D} \mid \theta) d\theta$ is not equal to one.

To show this, we approximate $\int_0^1 p(\mathcal{D} \mid \theta) d\theta$ numerically using the rectangle rule.

[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:.4}')
```

```
Integral = 3.068e-06
```

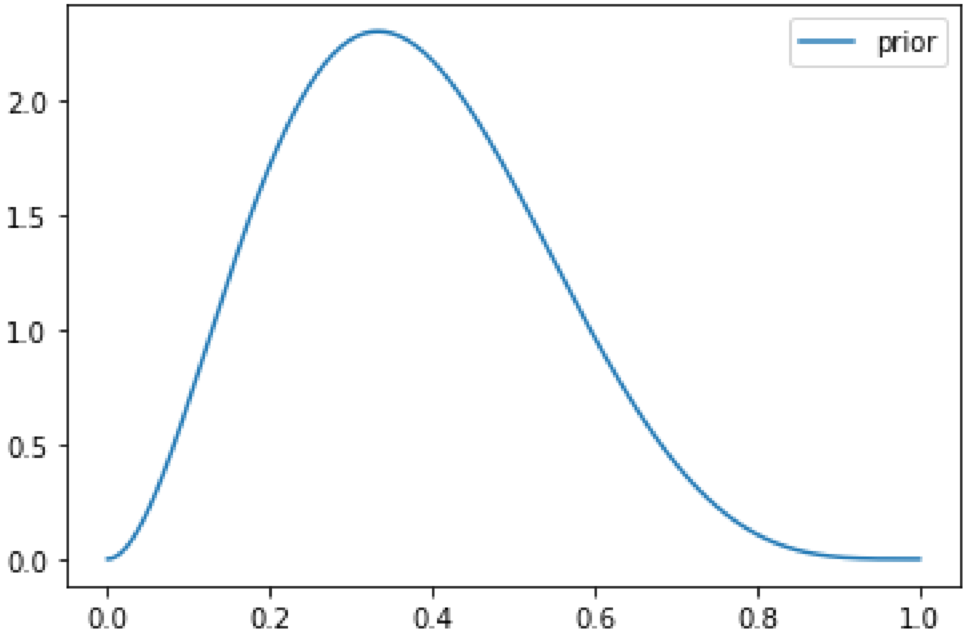
2.2 Task 2: Compute $\log p(\theta \mid a, b)$ for different values of θ

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

[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 = loggamma(a + b) - loggamma(a) - loggamma(b) + (a - 1) * np.
 ↪log(theta) + (b - 1) * np.log(1 - theta)
    return log_prior
```

[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()
```

[8]:
```
<matplotlib.legend.Legend at 0x7f837ab1a700>
```



Unlike the likelihood, the prior defines a probability distribution over θ and integrates to 1.

[9]:

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

```
Integral = 0.999
```

2.3 Task 3: Compute $\log p(\theta \mid \mathcal{D}, a, b)$ for different values of θ

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

[10]:

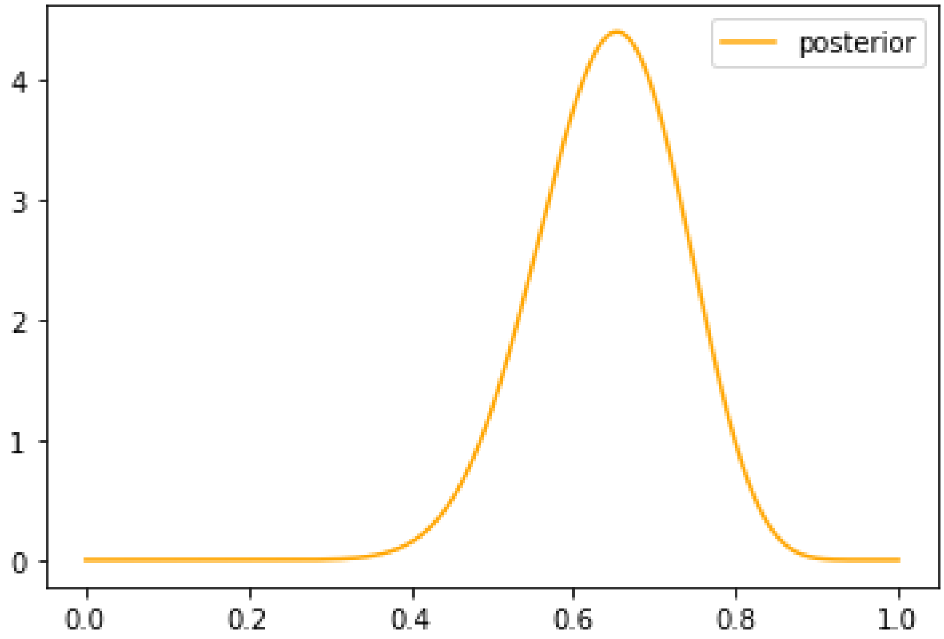
```
def compute_log_posterior(theta, samples, a, b):
    """Compute log p(theta | D, a, b) for the given values of theta.

    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 ###
    T, H = 0, 0
    for sample in samples:
        if sample == 1:
            T = T + 1
        else:
            H = H + 1
    log_posterior = loggamma(T + H + a + b) - loggamma(T + a) - loggamma(H + b)␣
 ↪+ (T + a - 1) * np.log(theta) + (H + b - 1) * np.log(1 - theta)
    return log_posterior
```

[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()
```

[11]:

```
<matplotlib.legend.Legend at 0x7f837a286c10>
```



Like the prior, the posterior defines a probability distribution over θ and integrates to 1.

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

```
Integral = 0.999
```

2.4 Task 4: Compute θ_{MLE}

```
[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 ###
          T, H = 0, 0
          for sample in samples:
```

```
        if sample == 1:
            T = T + 1
        else:
            H = H + 1
    return T / (T + H)
```

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

```
theta_mle = 0.750
```

2.5 Task 5: Compute θ_{MAP}

```
[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 ###
          T, H = 0, 0
          for sample in samples:
              if sample == 1:
                  T = T + 1
              else:
                  H = H + 1
          return (T + a - 1) / (T + H + a + b - 2)
```

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

```
theta_map = 0.654
```

3 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.

[17]:

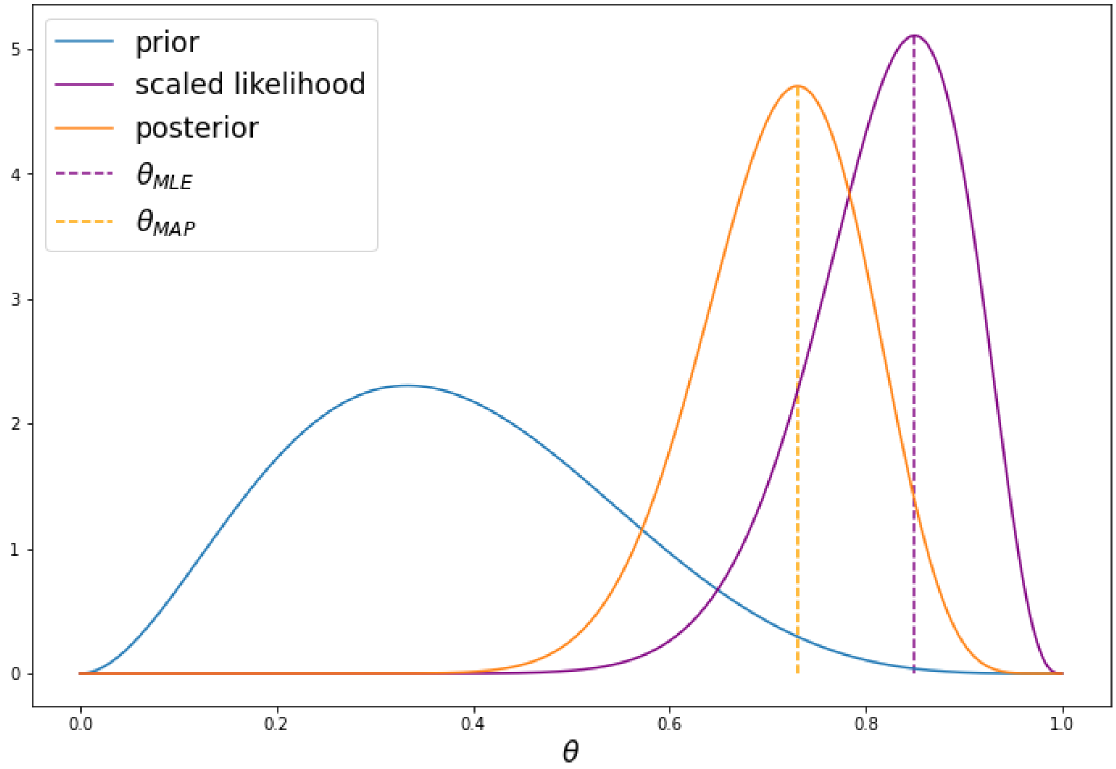
```
num_samples = 20
tails_proba = 0.7
samples = simulate_data(num_samples, tails_proba)
a, b = 3, 5
print(samples)
```

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

[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', label=r'$\theta_{MLE}$')


# 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', label=r'$\theta_{MAP}$')

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



```
[ ]:
```