exercise_03_notebook

October 26, 2019

1 Programming assignment 3: Probabilistic Inference

```
In [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 format of input and output. Write your code in a way that adheres to it. You may only use plain python and `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. 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 a Linux machine 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.

```
In [155]: # 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_pro
```

```
In [156]: 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).

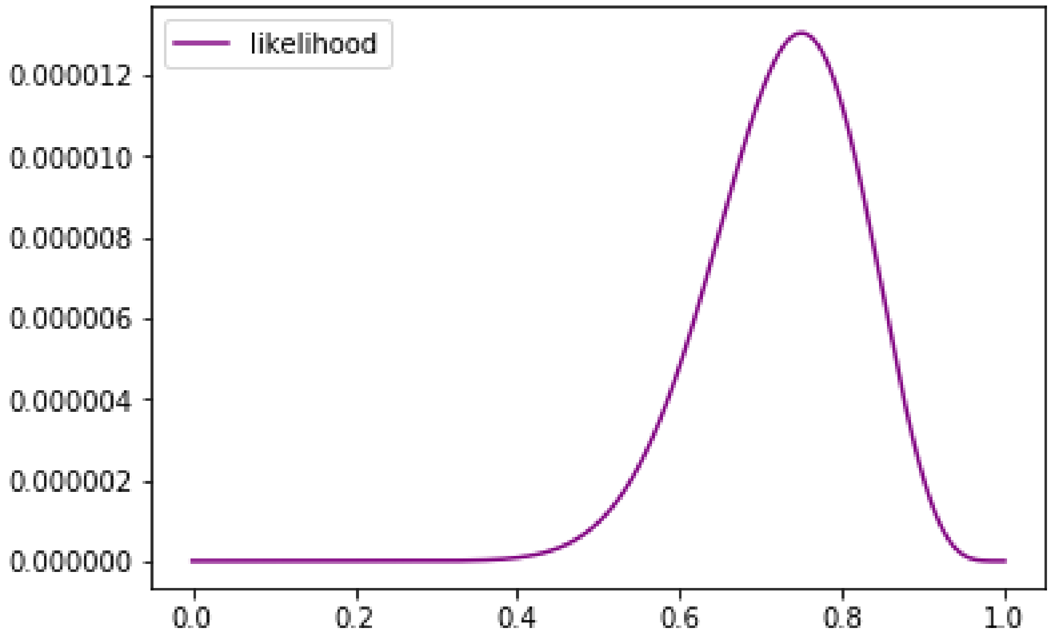
2.1 Task 1: Compute $\log p(\mathcal{D} \mid \theta)$ for different values of θ

```
In [157]: 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_samples = samples.shape[0]
    tails = sum(samples)
    heads = num_samples - tails

    # naiive
    #log_likelihood = np.log((theta**tails) * ((1-theta)**heads))

    # better (numerically more stable)
    log_likelihood = (tails * np.log(theta)) + (heads * (np.log(1-theta)))


    return log_likelihood
```

```
In [158]: 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[158]: <matplotlib.legend.Legend at 0x8240abf28>
```



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.

```
In [159]: # 1.0 is the length of the interval over which we are integrating p(D | theta)
          int_likelihood = 1.0 * np.mean(likelihood)
          # Excpeted output: Integral = 3.068e-06
          print(f'Integral = {int_likelihood:.4}')
```

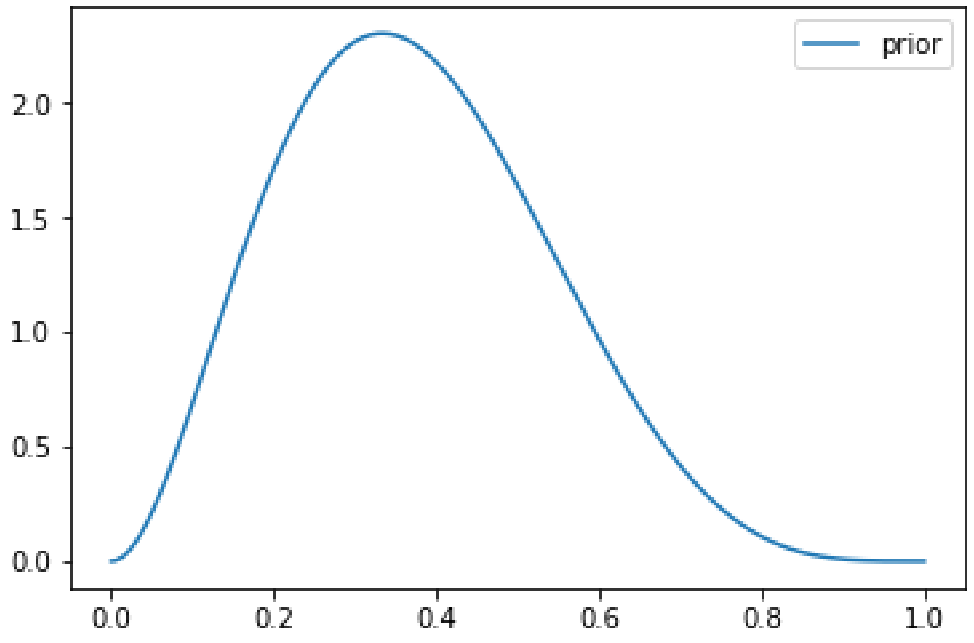
```
Integral = 3.068e-06
```

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)

```
In [160]: 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 ###

              # naiive
              #from scipy.special import gamma as G
              #gamma = G(a+b) / (G(a) * G(b))
              #log_prior = np.log(gamma * (theta**(a-1)) * ((1-theta)**(b-1)))

              # better (numerically more stable)
              gamma = loggamma(a+b) - (loggamma(a) + loggamma(b))
              log_prior = gamma + ((a-1) * np.log(theta)) + ((b-1) * np.log(1-theta))

              return log_prior
```

```
In [161]: 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[161]: <matplotlib.legend.Legend at 0x8241f3e48>
```



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

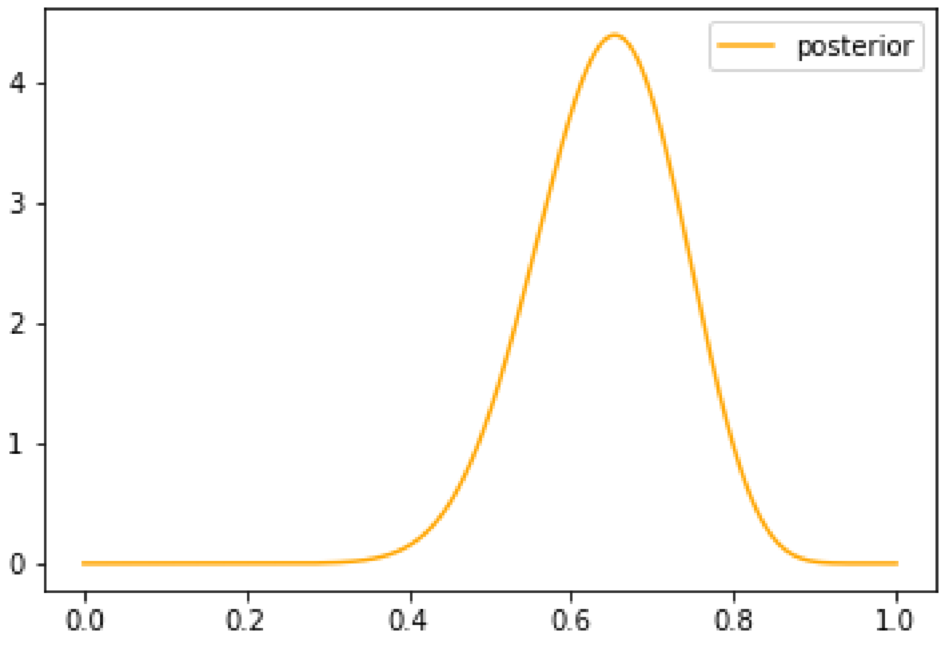
```
In [162]: int_prior = 1.0 * np.mean(prior)
          # Expected output: Integral = 0.999
          print(f'Integral = {int_prior:.4}')

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.

```
In [296]: 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 ###

            num_samples = samples.shape[0]
            tails = sum(samples)
            heads = num_samples - tails
            at = a + tails
            bh = b + heads

            # naiive (but did not quite work ...)
            #log_likelihood = compute_log_likelihood(theta, samples)
            #log_prior = compute_log_prior(theta, a, b)
            # Both did not work, why?
            #int_likelihood = np.mean(np.exp(log_likelihood)) * np.mean(np.exp(log_prior))
            #int_likelihood = (loggamma(at) + loggamma(bh)) - (loggamma(at + bh))
            #log_posterior = (log_likelihood + log_prior) - int_likelihood

            # slide 30
            gamma = loggamma(at+bh) - (loggamma(at) + loggamma(bh))
            log_posterior = gamma + ((at-1) * np.log(theta)) + ((bh-1) * np.log(1-theta))

            return log_posterior

In [297]: 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[297]: <matplotlib.legend.Legend at 0x82709bb00>
```



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

```
In [298]: int_posterior = 1.0 * np.mean(posterior)
          print(f'Integral = {int_posterior:.4}')

Integral = 0.999
```

2.4 Task 4: Compute θ_{MLE}

```
In [303]: 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 ###

              num_samples = samples.shape[0]
              tails = sum(samples)
```

```
        heads = num_samples - tails

        return tails / (tails + heads)
```

```
In [304]: theta_mle = compute_theta_mle(samples)
          # Expected output: theta_mle = 0.750
          print(f'theta_mle = {theta_mle:.3f}')
```

```
theta_mle = 0.750
```

2.5 Task 5: Compute θ_{MAP}

```
In [305]: 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_samples = samples.shape[0]
              tails = sum(samples)
              heads = num_samples - tails

              # slide 26
              return (tails + a - 1) / (heads + tails + a + b - 2)
```

```
In [306]: theta_map = compute_theta_map(samples, a, b)
          # Expected output: theta_map = 0.654
          print(f'theta_map = {theta_map:.3f}')
```

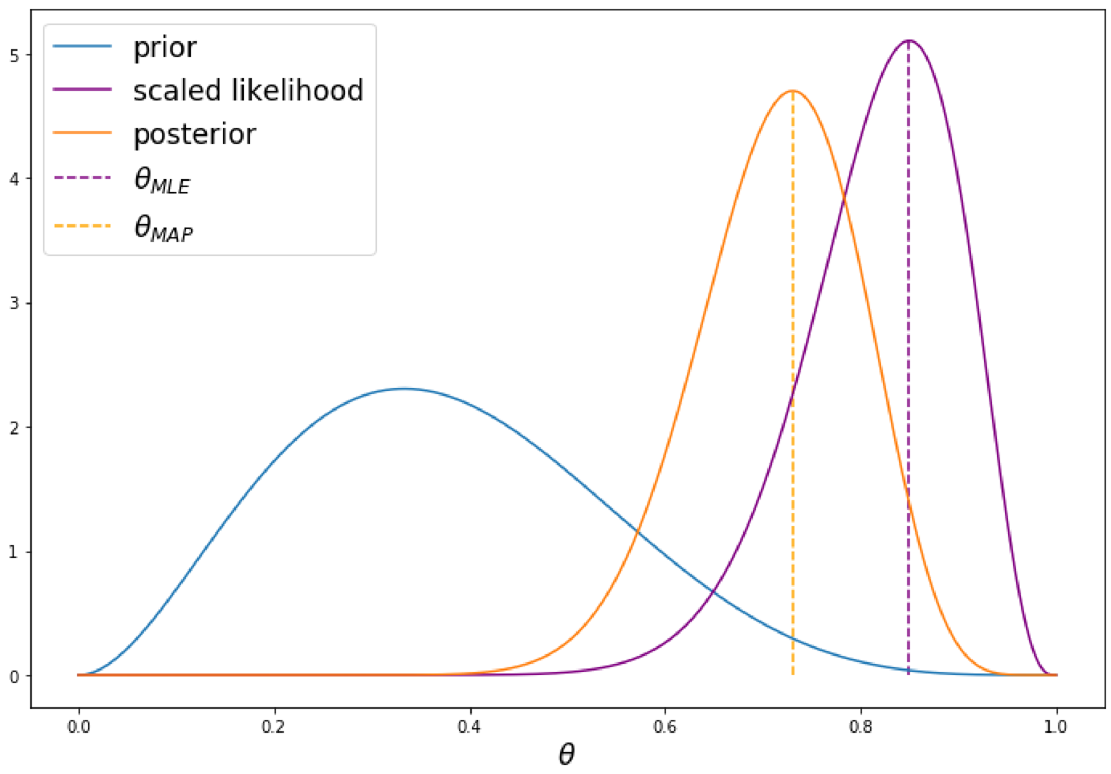
```
theta_map = 0.654
```

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.

```
In [307]: 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]
```

```
In [308]: 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_likeliho
          plt.vlines(x=theta_mle, ymin=0.00, ymax=ymax, linestyle='dashed', color='purple', lal


          # 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', lal

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



prior
scaled likelihood
posterior
θ_{MLE}
θ_{MAP}
θ

```
In [ ]:
```