Exercise

03

TUM Department of Informatics

Supervised by Prof. Dr. Stephan Günnemann

Informatics 3 - Professorship of Data Mining and Analytics

Submitted by Marcel Bruckner (03674122)

Julian Hohenadel (03673879)

Kevin Bein (03707775)

Submission date Munich, November 2, 2019

Optimizing Likelihoods: Monotonic Transforms

Problem 1:

$$f_1(\theta) = \theta^t(1-\theta)^h$$

$$f_2(\theta) = \log(\theta^t(1-\theta)^h)$$

$$\begin{aligned}
\frac{\partial}{\partial\theta}f_1(\theta) &= t\theta^{t-1}(1-\theta)^h + \theta^t\frac{\partial}{\partial\theta}\Big((1-\theta)^h\Big) \\
&= t\theta^{t-1}(1-\theta)^h - \theta^t h(1-\theta)^{h-1} \\
\frac{\partial^2}{\partial\theta}f_1(\theta) &= t\cdot\frac{\partial}{\partial\theta}\Big(\theta^{t-1}(1-\theta)^h\Big) - h\cdot\frac{\partial}{\partial\theta}\Big(\theta^t(1-\theta)^{h-1}\Big) \\
&= t\cdot\Big((t-1)\theta^{t-2}(1-\theta)^h - \theta^{t-1}h(1-\theta)^{h-1}\Big) \\
&+ h\cdot\Big(t\theta^{t-1}(1-\theta)^{h-1} - \theta^t(h-1)(1-\theta)^{h-2}\Big)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial\theta}f_2(\theta) &= \frac{\partial}{\partial\theta}\Big(\log(\theta^t) + \log((1-\theta)^h)\Big) \\
&= \frac{\partial}{\partial\theta}\left(\log(\theta^t)\right) + \frac{\partial}{\partial\theta}\Big(\log((1-\theta)^h)\Big) \\
&= t\cdot\frac{\partial}{\partial\theta}(\theta) + h\cdot\frac{\partial}{\partial\theta}\left(\log(1-\theta)\right) \\
&= \frac{t}{\theta} - \frac{h}{1-\theta}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2}{\partial\theta}f_2(\theta) &= \frac{\partial}{\partial\theta}\left(\frac{t}{\theta}\right) - \frac{\partial}{\partial\theta}\left(\frac{h}{1-\theta}\right) \\
&= \frac{0\cdot\theta - t\cdot 1}{\theta^2} - \frac{0\cdot(1-\theta) - h\cdot(-1)}{(1-\theta)^2} \\
&= -\frac{t}{\theta^2} - \frac{h}{(1-\theta)^2}
\end{aligned}$$

Problem 2:

$$\frac{\partial}{\partial \theta} \log f(\theta) = \frac{1}{f(\theta)} * \frac{\partial}{\partial \theta} f(\theta) \stackrel{!}{=} 0 \stackrel{!}{=} \frac{\partial}{\partial \theta} f(\theta)$$

The derivative of the logarithm of a function f is the same as the derivative of the function times the factor $\frac{1}{f}$. So in all minima and maxima where the derivative of the function f is 0, the derivative of the function times a factor is also 0, hence the logarithm keeps all minima and maxima.

Since $\log$ is a monotonic transformation, the function will yield the same values when plugged into $\arg\max_{\theta}$:

$$\arg\max_{\theta} f(\theta) = \arg\max_{\theta} \log f(\theta)$$

Considering problem 1 and the fact that log converts exponents to factors (which also increases the numerical stabillity), computing the log-likelihood is faster and yields more compact functions which can then be maximized.

Properties of MLE and MAP

Problem 3:

$$\begin{aligned}
Beta(6,4) &= \left(\frac{\Gamma(6+4)}{\Gamma(6)\Gamma(4)} \cdot \theta^{6-1} \cdot (1-\theta)^{4-1}\right) \\
&= \frac{9!}{5! \cdot 4!} \cdot \theta^5 \cdot (1-\theta)^3 \\
&= 126 \cdot \theta^5 \cdot (1-\theta)^3
\end{aligned}$$

$$\begin{aligned}
\theta_{\text{MAP}} &= \arg\max_\theta p(\theta|f) \\
&= \arg\max_\theta \frac{p(f|\theta) \cdot p(\theta)}{p(f)} \\
&= \arg\max_\theta p(f|\theta) \cdot p(\theta) \\
&= \arg\max_\theta \left(\theta^{\mathbb{I}[f=T]} \cdot (1-\theta)^{\mathbb{I}[f=H]}\right) \cdot \left(126 \cdot \theta^5 \cdot (1-\theta)^3\right) \\
&= \arg\max_\theta \theta^{M+5} \cdot (1-\theta)^{N+3} \\
&= \arg\max_\theta \log\left(\theta^{M+5} \cdot (1-\theta)^{N+3}\right) \\
&= \arg\max_\theta (M+5)\log(\theta) + (N+3)\log(1-\theta)
\end{aligned}$$

$$\begin{aligned}
&\Rightarrow \frac{\partial}{\partial\theta}(M+5)\log(\theta) + (N+3)\log(1-\theta) = \frac{M+5}{\theta} - \frac{N-3}{1-\theta} \stackrel{!}{=} 0 \\
&\Rightarrow \theta_{\text{MAP}} = \frac{M+5}{M+N+8}
\end{aligned}$$

$$\begin{aligned}
\theta_{\text{MAP}} &\stackrel{!}{=} 0.75 = \frac{3}{4} = \frac{30}{40} \\
\Rightarrow\ M &= 25, N = 40 - 25 - 8 = 7
\end{aligned}$$

(In general, the solution is $M = 3N + 1$)

Problem 4:

Given are $X \sim Bern$, $P(\theta) \sim Beta$ and $P(D|\theta) \sim Binomial$. The expecterd mean of the prior (Beta distribution) is $\mathbb{E}[\theta] = \frac{a}{a+b}$. The maximum likelihood estimate is given by $\mathbb{E}[\theta_{\text{MLE}}] = \frac{m}{N} = \frac{m}{m+l}$. We now identify the posterior mean distribution:

$$\begin{aligned}
P(\theta|D) &= \frac{P(D|\theta)P(\theta)}{P(D)} \\
&= \left(\binom{N}{m}\theta^m(1-\theta)^{N-m}\right) \cdot \left(\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\theta^{a-1}(1-\theta)^{b-1}\right) \\
&= \binom{N}{m}\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \cdot \theta^{m+a-1} \cdot (1-\theta)^{N-m+b-1} \\
&\propto \theta^{m+a-1} \cdot (1-\theta)^{N-m+b-1} \\
&\propto Beta(\theta|m+a, N-m+b) = Beta(\theta|m+a, l+b)
\end{aligned}$$

Looking up the mean of the Beta distribution yields the mean of the posterior distribution:

$$\mathbb{E}[\theta|D] = \mathbb{E}[\theta|m, N, a, b] = \frac{m+a}{m+a+N-m+b} = \frac{a+m}{a+b+m+l}$$

Refactoring the mean

$$\begin{aligned}
\frac{a+m}{a+b+m+l} &= \frac{a}{a+b+m+l} + \frac{m}{a+b+m+l} \\
&= \underbrace{\frac{a+b}{a+b+m+l}}_{\lambda} \cdot \frac{a}{a+b} + \underbrace{\frac{m+l}{a+b+m+l}}_{(1-\lambda)} \cdot \frac{a}{m+l} \\
&= \lambda \cdot \frac{a}{a+b} + (1-\lambda) \cdot \frac{m}{m+l} \\
&= \lambda \cdot \mathbb{E}[\theta] + (1-\lambda) \cdot \mathbb{E}[\theta_{\text{MLE}}]
\end{aligned}$$

shows that the posterior mean lies between the prior mean and the maximum likelihood estimate for θ.

Programming Task

Problem 5:

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).

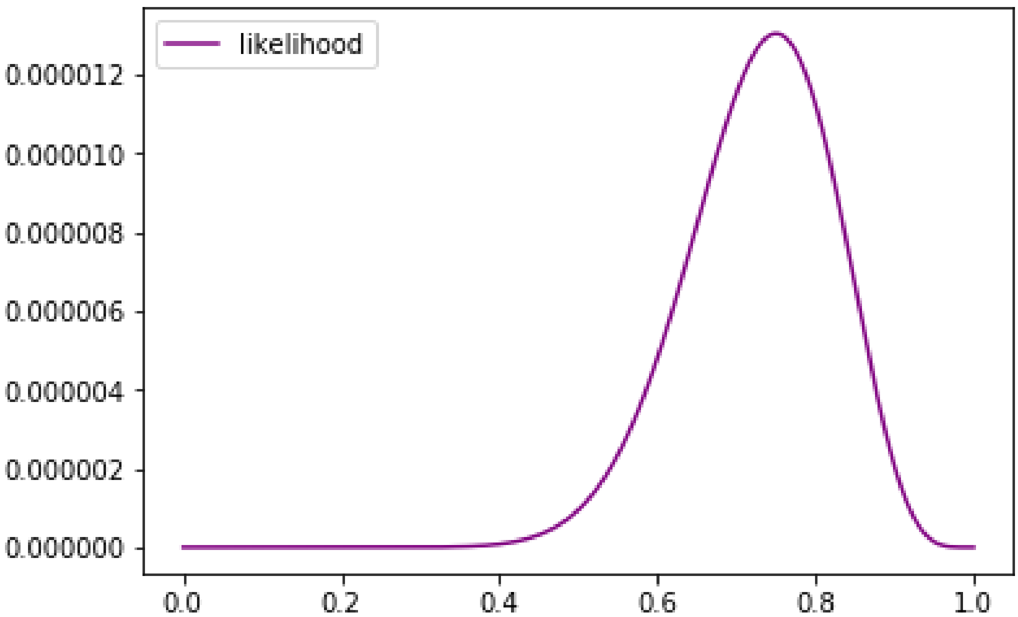
2.1 Task 1: Compute $\log p(\mathcal{D} \mid \theta)$ for different values of θ

```
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

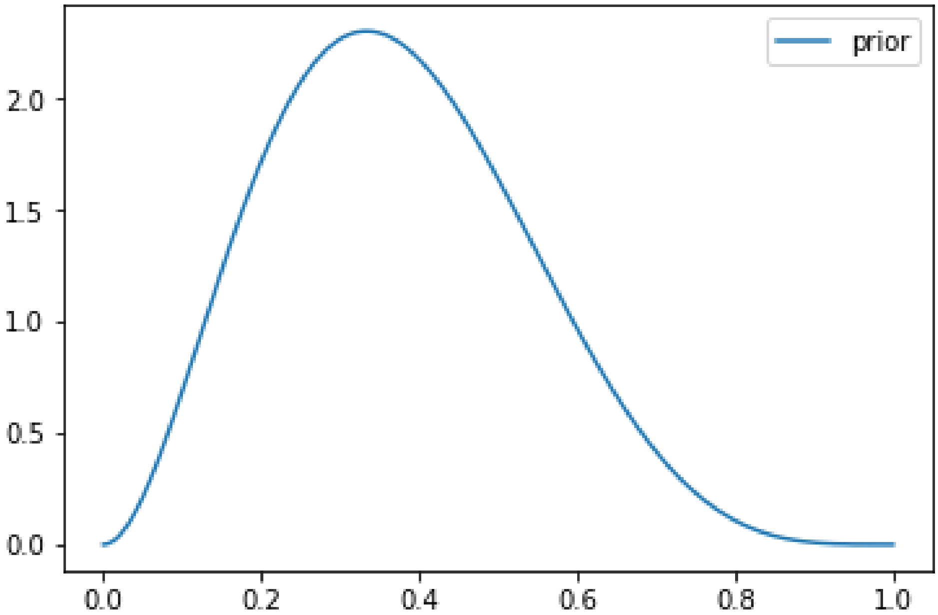
The function `loggamma` from the `scipy.special` package might be useful here. (It's already imported - see the first cell)

```
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
```

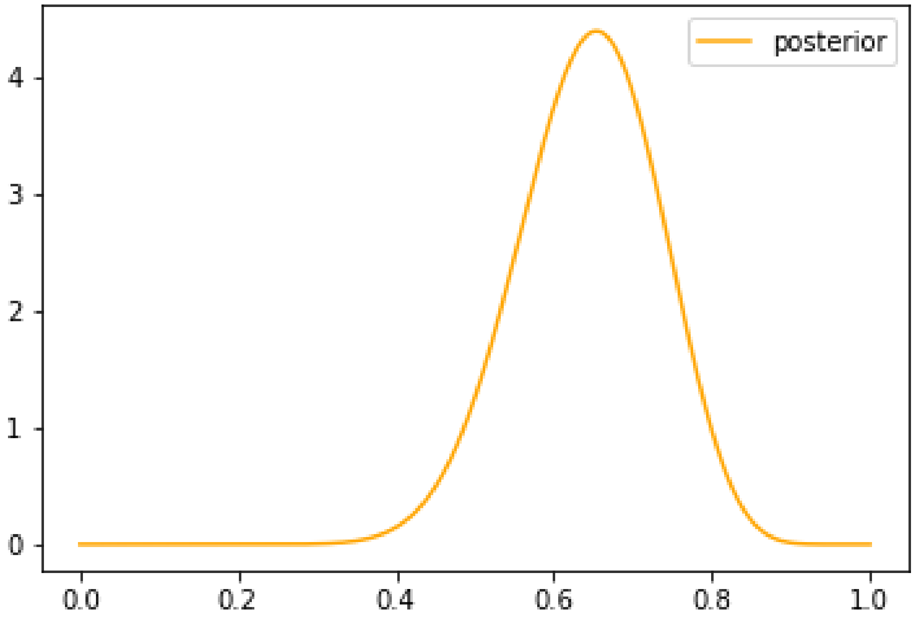
2.3 Task 3: Compute $\log p(\theta \mid \mathcal{D}, a, b)$ for different values of θ

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
```

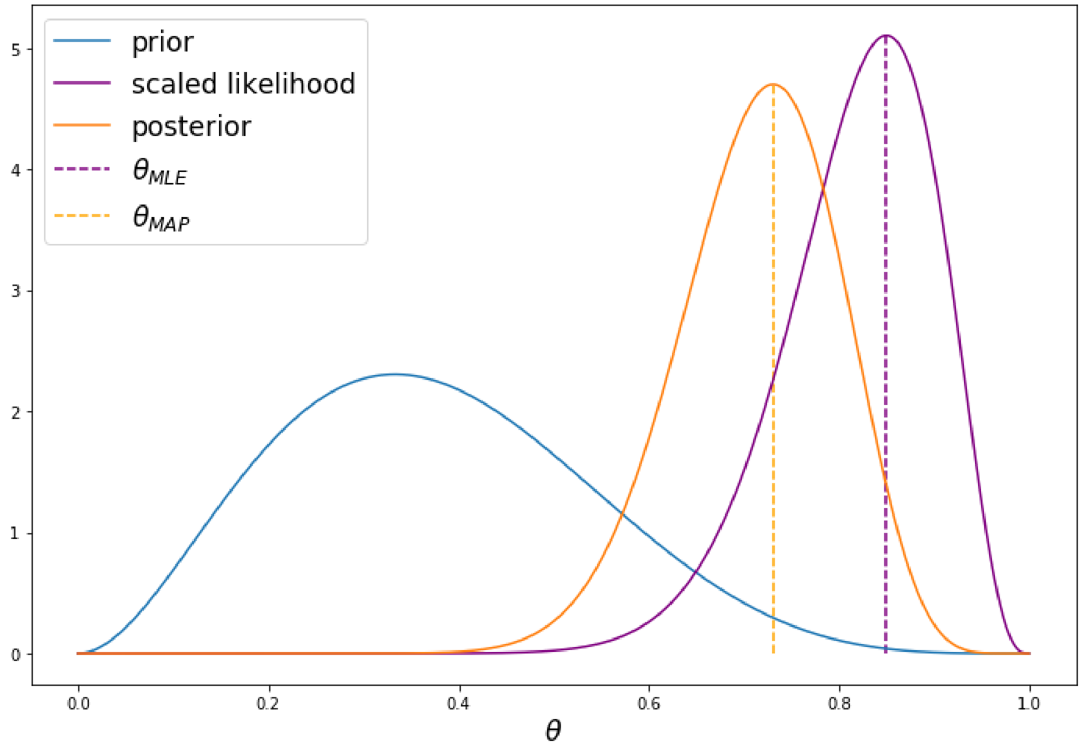
3 Putting everything together

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}
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0
θ


```
In [ ]:
```

Appendix

We confirm that the submitted solution is original work and was written by us without further assistance. Appropriate credit has been given where reference has been made to the work of others.

Munich, November 2, 2019, Signature Marcel Bruckner (03674122)

Munich, November 2, 2019, Signature Julian Hohenadel (03673879)

Munich, November 2, 2019, Signature Kevin Bein (03707775)