Exercise Sheet 3
Markov Chain Monte Carlo

Solutions. You do not need to submit your exercise solutions to us. Of course you are free to ask questions during tutorial hours! Exercise solutions will be published on Moodle at the same time with the exercise sheet. Use solutions responsibly, otherwise you risk not being able to solve the exam to a satisfactory level.

Exercise 1 Single-chain Metropolis algorithm from scratch

The Metropolis algorithm is a very good start into the world of Markov Chain Monte Carlo. Even though it is less effective than Hamiltonian MCMC or NUTS, its simplicity allows easy investigation of the principles of MCMC algorithms. In this exercise you will build the Metropolis algorithm from scratch and apply it to a problem with binomial likelihood. We'll still use the Beta prior from last week, but pretend we don't know the analytical solution yet (of course we'll use it to verify whether our implementation works correctly so that later we can work with arbitrary priors).

Problem to solve:

Given $\alpha = 5$, $\beta = 2$, $n = 10$ and $k = 8$:

- $\pi \sim \text{Beta}(\alpha, \beta)$ (prior)
- $Y{=}k|\pi \sim \text{Bin}(n, \pi)$ (likelihood)
- $\pi|Y{=}k \sim ?$ (posterior)

a) **Proposal distribution**

You will use a Gaussian jump proposal distribution for this particular implementation. Implement a Python function `propose_jump(pi, sigma)`, where `pi` is the current value in the chain and `sigma` is the standard deviation of the proposal distribution. Use `scipy.stats.norm.rvs()` to propose a new value $\pi' \sim N(\pi, \sigma^2)$ that is returned by the function.

b) **Random walk**

Create a simple (Markovian) Monte Carlo chain using your `propose_jump()` function. Initialize your chain with a random value for π from the interval $[0, 1]$ and accept all proposed jumps. Do this for 1000 iterations with a standard deviation of `sigma`=0.2. Plot the values of the resulting chain (random walk). Run your code multiple times and look at the different random walks you produced.

Hint: Make sure you clip the proposed values to a range between 0 and 1. The proposal distribution is not aware of the fact that π is bounded between 0 and 1.

c) **Jump decision**

In the Metropolis algorithm (assuming a symmetric proposal function), the classic random walk is nudged towards our posterior by applying a decision rule for whether a proposed jump is accepted. This rule is the following:

Given are the current value π in the chain and the proposed value π'. Jump to π' with the probability

$$\alpha = \min\left\{1, \frac{p(\pi')\,P(Y=k|\pi')}{p(\pi)\,P(Y=k|\pi)}\right\}.$$

Once you have computed α (Hint: use `stats.beta.pdf()` and `stats.binom.pmf()`), you may sample a value $p \sim U(0, 1)$ (using `np.random.rand()`) and use the following decision rule:

- If $p \leq \alpha$: return the proposed value π' (accept)
- If $p > \alpha$: return the original value π (reject)

Write a function `decide(pi, pi2, alpha, beta, n, k)`, where `pi` is the current value in the chain, `pi2` is the proposed value by `propose_jump(pi, sigma)`, `alpha` and `beta` define the beta prior and `n` and `k` describe the measured binomially distributed data. Test your function by definining a fixed value for `pi`, proposing a jump `pi2` with your `propose_jump()` function, run it through `decide()` and verify that it sometimes returns `pi` and sometimes the proposed value.

Hints:

- You may print out or return α to verify whether your code works correctly.
- The jump proposal function might propose a value of π outside of its domain $[0, 1]$. Make sure that in that case the proposal is rejected.

d) **Single-chain Metropolis algorithm**

Now it's time to put everything together to have your own implementation of the Metropolis algorithm! Create an empty list called `chain`, randomly sample an initial value for π between 0 and 1 and use a for loop to iteratively apply your `propose_jump()` and `decide()` functions `n_steps` times. Make sure you store the current value of `pi` in `chain` in each step.

Encapsulate your code in a function `metropolis_chain(n_steps, alpha, beta, n, k, sigma)` that returns the final chain as a list with `n_step` entries. Generate a chain with 1000 steps (samples).

e) **Verification**

You may now verify whether your `metropolis_chain()` function works correctly. To this end, perform the following steps:

1. **Hyperparameter tuning:** Visualize your chain in a **trace plot** by simply plotting the evolution of your chain values. Does the trace plot look alright? If the chain is mixing slowly (acceptance rate too high), increase `sigma`, if the chain is often stuck at a value (acceptance rate too low), decrease `sigma`. Before you continue, make sure that your trace plot looks ok.
2. **Visualize distribution:** Now visualize the distribution of your samples using a histogram. Thereby make sure, that your histogram shows a density instead of absolute counts. Overplot the density with the PMF of a Beta$(\alpha + k, \beta + n - k)$ distribution (as we computed it in closed form last week). Does the distribution of the samples more or less match the closed-form distribution?

f) **Playing around**

If you have not done this yet: Play around with different values for the proposal jump distribution standard deviation `sigma`. What is the effect on the trace plot and the histogram (see previous part) if you set `sigma` too small, too high or just right? What tells you that you chose `sigma` just right? (apart from the true closed-form posterior distribution).

Exercise 2 Multi-chain Metropolis algorithm

In the previous exercise you used the trace plot as a diagnostic for whether your chain is healthy (fast mixing). Here we investigate another instrument: density plots of multiple chains.

a) **Multiple chains**

Write a function `metropolis(n_chains, n_steps, alpha, beta, n, k, sigma)` that runs `n_chains` different chains (let's start with 4) using your `metropolis_chain()` function from the previous exercise. Your function should return a numpy array of dimension (n_steps, n_chains) that you can then use in the remaining parts of the exercise.

b) **Trace plots**

Visualize all four trace plots of your chains, use your best value for `sigma` from the previous exercise. Does everything look ok?

c) **Density plots**

Visualize all four chains in separate density plots (use e.g. `seaborn.kde_plot()`). Are they more or less in agreement? What happens with them if you set `sigma` too large or too small?

d) $\hat{\mathbf{R}}$

In the lecture we looked at the quantitative measure $\hat{R}$ to indicate how well the four chains converged towards each other. Compute

$$\hat{R} = \sqrt{\frac{\mathrm{Var}(\mathbf{x})}{\frac{1}{n_{\text{chains}}}\sum_i \mathrm{Var}(\mathbf{x}_i)}}$$

for your output, where $\mathbf{x}_i$ is the list of values in chain i and $\mathbf{x}$ is the concatenation of all $\mathbf{x}_i$. What's the value of $\hat{R}$ for your optimal `sigma`? How does it change if you make `sigma` too small or too high? Is $\hat{R}$ enough to distinguish good from bad chains?

e) **Burn-in phase**

Is the burn-in phase typically required by the Metropolis algorithm relevant for your beta-binomial problem? Argue using your trace plots.

Exercise 3 Doing the same with PyMC

a) Reproduce the results from the previous exercise with PyMC (see lecture notes) to demonstrate that PyMC does almost everything for you, except from specifying prior and likelihood of course.

b) Investigate the output of `pm.sample()`. How can you access the samples for π?

c) Produce the following diagnostic artefacts and check whether they look alright (using the PyMC-internal functions or the arviz equivalents):

   - trace plot + density plot
   - autocorrelation plot
   - effective sample size (ESS)
   - $\hat{R}$

   Finally, if you have graphviz installed, use `pm.model_to_graphviz()` to produce a graphical representation of your very simple model. What is the meaning of the arrow?

d) Some practitioners argue that a rank plot (or trank plot) is better suited to spot bad chains than a trace plot. Investigate what a rank plot is and find out how you can produce it with PyMC. Does it look good for your simulation?

e) A coworker of yours argues that a truncated normal distribution with $\mu = 0.8$ and $\sigma = 0.2$, truncated between 0 and 1, would describe his prior belief better. Introduce it to your simulation and argue that it does not make a big difference which of these two priors you use. In particular,

   1. do some research to find out what a truncated normal distribution is,
   2. use preliz to visualize your beta prior and your coworker's truncated normal prior and quickly compare them visually,
   3. run a new simulation using your coworker's prior and compare the density plots of both simulations,
   4. use PyMC's `summary()` function to give your coworker a quick comparison. Does it matter a lot which of the two priors you use? What about bulk ESS and tail ESS? (what are they, what does it mean when they are different?)

Exercise 4 Foundry

A company produces containers for foundries ('Giessereien') using ceramics and wants to know the maximum temperature such a container can tolerate and in particular also the uncertainty on it. Since finding out this maximum temperature destroys a container, as few measurements as possible should be made.

Your team has started with 10 measurements and wants to know from you (as data scientist) to what accuracy you can make a statement about the average maximum temperature and its uncertainty.

The measurements are (in degrees Celsius)

$$\mathbf{y} = [1055, 1053, 1226, 967, 980, 1049, 1040, 1051, 1002, 1057]$$

and you consider the following model:

$$\begin{aligned} \mu &\sim N(\theta, \tau^2), \\ \sigma &\sim \text{Exp}(1/\lambda), \\ y|\mu, \sigma &\sim N(\mu, \sigma^2). \end{aligned}$$

You have a prior expectation about μ (prior normally distributed) and σ (prior exponentially distributed) with $\theta = 1000°$ C, $\tau = 100°$ C and $\lambda = 100°$ C and assume a normal likelihood for your data. This is the first time that you see a model where we want to estimate multiple parameters instead of just one. Instead of defining one combined two-dimensional prior for μ and σ, we assume independence and define two individual priors. We will discuss this better in the lecture in week 5.

a) Read about the exponential distribution. What is the role of λ? Does it favour lower or higher values than λ? Visualize Exp(1/100) with preliz. Why would a normal distribution or a beta distribution not be a good prior for σ?

b) Carefully formulate the simulation with PyMC and run it. Verify the convergence of your chain using a trank plot and compute posterior summaries with `pm.summary()`. What are your posterior expectations for μ and σ and how do they compare to your prior expectations?

c) If you have graphviz installed: Use `pm.model_to_graphviz()` to visualize your model. Can you relate it to the model description above?

d) Extract the first 100 samples for μ and σ from the first chain and visualize them against each other in a two-dimensional plot that shows the evolution of your chain. Can you see evidence for a warm-up phase? Note: Advanced MCMC algorithms do not have a burn-in phase (finding the mass of the posterior), they have a **warm-up** phase (automatically adjusting the hyperparameters).