Fifth Lab Session: Statistical uncertainty

UE Computational Statistics

moi

Part 0. Installations

We will need the following packages:

- JAX: a numerical computing library for Python that is composable, fast, and differentiable
- PyMC: a python library for probabilistic programming
- Arviz: a python library for exploratory analysis of Bayesian models

In an Anaconda Terminal/Console/Command Prompt, run the following:

1. Create a new environment with `conda create -c conda-forge -n comp_stat "pymc>=5"`
2. Activate the environment with `conda activate comp_stat`
3. Install the packages with `conda install -c conda-forge jax jaxlib arviz ipykernel ipywidgets python-graphviz`
4. Check that our favorite packages are installed with `conda install numpy scipy matplotlib seaborn pandas`
5. (Optionnal) If you want PyMC to be efficient, you can also install numpyro with `conda install -c conda-forge numpyro`
6. Finally, in the Jupyter Notebook, be sure to select the kernel `comp_stat` to run the code.

```python
import numpy as np
import scipy
import scipy.stats as stats
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import jax.numpy as jnp
from jax import grad, hessian, jit
sns.set_style("whitegrid")
```

Part 1. Logistic regression

1.1 The statistical model

The statistical model of the logistic regression is as follows. First, the random datset is:

$$D = \begin{pmatrix}\mathbf{Y} & \mathbf{X}\end{pmatrix} = \begin{pmatrix} Y_1 & X_{11} & \cdots & X_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ Y_n & X_{n1} & \cdots & X_{np} \end{pmatrix}$$

where $Y_i \in \{0, 1\}$ is the binary response variable, and X_{ij} are the j-th covariate of the i-th observation. The rows of D are iid, the marginal distribution of $X_i = (X_{i1}, \dots, X_{ip})$ is not specified, and the conditional distribution of Y_i given $(X_{i1}, \dots, X_{ip})$ is a Bernoulli distribution with parameter $p(X_i)$:

$$[Y_i|X_i] \sim \text{Bernoulli}\Big(p(X_i)\Big), \quad \text{with } p(X_i) = \frac{1}{1 + \exp\left(-\beta_0 - \sum_j \beta_j X_{ij}\right)}.$$

The (conditional) likelihood of the data d is:

$$f(\mathbf{y}|\beta, \mathbf{x}) = \prod_{i=1}^{n} p(x_i)^{y_i} \left(1 - p(x_i)\right)^{1-y_i}.$$

1.2 The data

The dataset is available in the `wells.csv` file. We can import it with the following code:

```python
data = pd.read_csv('wells.csv')
data.head()
```

This example is taken from the book by Gelman, Hill, and Vehtari (2020). The wells used by the inhabitants of Bangladesh are contaminated with natural arsenic, as in other South Asian countries. Arsenic is a poison whose risks accumulate proportionally to the duration and dose of exposure. A well is considered safe when the arsenic dose it contains is less than 0.5 (in hundreds of micrograms per liter). The wells are located in living areas. When a well is not safe, it is very common for a user to find a safe well in their neighborhood without over-exploiting and drying it up because the potable water needed for human consumption represents a small volume.

The data studied comes from the work of a research team from the USA and Bangladesh on wells in the Araihazar region. This team measured the arsenic level of all wells in the region and labeled them as "safe" or "unsafe". Households that were using unsafe wells were encouraged to switch wells. A few years later, the research team returned to the field to find out which households had (or had not) switched wells. The observations in the dataset correspond to different households in this region. The variables are:

- `switch`: 1 if the household has switched wells, 0 otherwise
- `arsenic`: the level of arsenic (in hundreds of micrograms per liter)
- `assoc`: 1 if a household member is active in community organizations, 0 otherwise
- `dist`: the distance in meters to the nearest safe well
- `educ`: education level of the household head

Note that two other variables are in the dataset: `educ4` which is mainly `educ/4` and `dist100` which is mainly `dist/100`.

The response variable is `switch`. To set the covariates, we standardize the variables `arsenic`, `assoc`, `dist`, and `educ` to have mean 0 and standard deviation 1.

```
y = jnp.array(data['switch'])
x = jnp.array(data[['arsenic', 'assoc', 'dist', 'educ']])
x = (x - x.mean(axis=0)) / x.std(axis=0)
x.shape, y.shape
```

And we add a column of ones to the covariate matrix to account for the intercept.

```
x1 = jnp.hstack((jnp.ones((x.shape[0], 1)), x))
x1.shape
```

A first study of the data has been done with `statsmodels`. We want to recover these results in the following sections. The output of the analysis was:

Dep. Variable:	y	No. Observations:	3020
Model:	GLM	Df Residuals:	3015
Model Family:	Binomial	Df Model:	4
Link Function:	Logit	Scale:	1.0000
Method:	IRLS	Log-Likelihood:	-1953.9
Date:	Wed, 05 Mar 2025	Deviance:	3907.8
Time:	10:22:22	Pearson chi2:	3.05e+03
No. Iterations:	4	Pseudo R-squ. (CS):	0.06726
Covariance Type:	nonrobust		

	coef	std err	z	P>|z|	[0.025	0.975]
const	0.3364	0.038	8.740	0.000	0.261	0.412
x1	0.5171	0.046	11.226	0.000	0.427	0.607
x2	-0.0614	0.038	-1.615	0.106	-0.136	0.013
x3	-0.3448	0.040	-8.569	0.000	-0.424	-0.266
x4	0.1705	0.039	4.427	0.000	0.095	0.246

1.3 Fisher information

Write the log-likelihood function of the logistic regression model, using `jnp` functions. The input should be:

- `beta`: the parameter vector of the logistic regression model,
- `x`: the covariate matrix (including a first column of ones for the intercept),
- `y`: the response vector.

To this aim, you can use `jnp.dot` for matrix product, `jnp.exp`, `jnp.log`, and `jnp.sum`. No loop (explicit or implicit) is needed. To accelerate the computation, you can use the `@jit` decorator before the function definition.

The `@jit` decorator is used to ask compilation of the function by JAX (JIT stands for **Just-In-Time compilation**). This is a way to speed up the computation. The first time the function is called, JAX compiles the function and stores the compiled version. The next time the function is called, JAX uses the compiled version. The compilation time is not negligible, but the computation time is much faster. The compilation time is negligible when the function is called many times, which will be the case of the log-likelihood function.

```
# answer here
```

Using **automatic differentiation** in JAX, define an object that computes the gradient of the log-likelihood function with respect to the parameter vector `beta`, that is to say the score function. Do the same for the Hessian matrix. The input should be the same as for the log-likelihood function.

```
# answer here
```

Remark: In JAX, to use automatic differentiation, you need a pure function. That is why we need to pass `x1` and `y` as arguments of the log-likelihood function.

Write a python function that takes as input:

- `init`: the initial value of the parameter vector,
- `n_iter`: the number of iterations of the gradient descent algorithm,
- `step_size`: the factor in front of the step size that is proportional to $1/\sqrt{t+1}$,
- `x`: the covariate matrix (including a first column of ones for the intercept),
- `y`: the response vector,

and returns the maximum likelihood estimator `beta_MLE` of the logistic regression model. The function implements a gradient descent algorithm. (Warning: we want to **maximize** the log-likelihood function, not minimize it. Be sure to climb the hill, not to go down the hill.)

We want to accelerate the computations wit JIT compilation. Yet we have to tell JIT that the `n_iter` argument is static, that is to say that it does not change along the iterations. Otherwise we will not be able to use it to control the loop. To this aim, we use the `partial` function from the `functools` package. For example, if your inputs are in the same order as above, the decorator should be: `@partial(jit, static_argnums=(1,))`. To use this decorator, you have to import it with `from functools import partial`.

```
# answer here
```

Run the algorithm on `x1` and `y` defined above. The initial value of the parameter vector is `jnp.zeros(x.shape[1])`. The step size should be `0.005/np.sqrt(t+1)` where `t` is the iteration number. The algorithm stops after 200 iterations. Compare your results with the one from `statsmodels` above.

```
# answer here
```

Compute the observed information matrix at the MLE and use it to get the standard errors of each coordinate of the MLE. Compare your results with the one from `statsmodels` above. You can use `jnp.diag` and `jnp.linalg.inv`.

```
# answer here
```

Compare your results with the one from `statsmodels` above.

Final remark: to be sure that the output of the MLE function is the global maximum of the log-likelihood function, you can run the algorithm several times with different initial values of the parameter vector. At the end, we keep the best result.

1.4 The Bootstrap

Write a python function that takes as input:

- `x`: the covariate matrix (including a first column of ones for the intercept),
- `y`: the response vector,
- `B`: the number of bootstrap samples,

and return B bootstrap estimates of β for the logistic regression model. To sample individuals with replacement, i.e. numbers between 0 and $n-1$, you can use `np.random.choice(n, n, replace=True)`. We will not use JAX nor JIT for this function. There is two good reasons for that: (1) using RNG in JAX is difficult, (2) we need to collect the B estimates in a loop, which is difficult with JAX/JIT. See next section for a JAX implementation.

```
# answer here
```

Run the algorithm on `x1` and `y` defined above with `B=1000` and compute the standard deviation of each coordinate of the bootstrap estimates. Compare your results with the one from the observed information matrix above.

```
# answer here
```

Plot the histograms of the bootstrap estimates of each coordinate of β. Add the bell curve of the normal distribution with mean the MLE and standard deviation the standard errors of the MLE to each plot.

```
# answer here
```

1.5 Using RNG with JAX

Using RNG in JAX is not straightforward. The main difficulty is that the state of the RNG is not a hidden variable that is automatically updated when you use the RNG. In JAX, the state of the RNG is named a key. We have to explicitly:

- create a RNG key,
- pass it as an argument to the function that uses the RNG,
- update the key each time it is useful.

Here are a few examples:

```
import jax
key = jax.random.key(1234) # create a RNG key with seed=1234
x=[]
for i in range(10):
    # update the key, create a subkey to use the RNG
    key, subkey = jax.random.split(key)
    # then use it!
    x.append(jax.random.normal(subkey, (1, 2)))
xnp = jnp.vstack(x)
print(xnp)
```

We could also have created the RNG keys outside the loop:

```
key = jax.random.key(1234) # create a RNG key with seed=1234
keys = jax.random.split(key, 10) # create an array of 10 subkeys
x=[]
for i in range(10):
    x.append(jax.random.normal(keys[i], (1, 2)))
xnp = jnp.vstack(x)
print(xnp)
```

1.6 The JAX Bootstrap

Now that we have understood (a bit) how to use the RNG in JAX, we are in a better position to implement the bootstrap with JAX. Yet we need to deal with the loop part of the algorithm. The idea is to implement the body of the loop in a specific function.

```
@jit
def one_bootstrap(rng_key, x, y):
    n = x.shape[0]
    idx = jax.random.choice(rng_key, n, (n,), replace=True)
    return MLE(jnp.zeros(x1.shape[1]), 200, 0.005, x[idx], y[idx])
```

Then, all we need is to **vectorize** the function `one_bootstrap` with the `vmap` function of JAX. The idea is that the new function will take an array of `rng_key`'s and apply the function `one_bootstrap` to each element of the array. This will create the loop.

Note that we want `x` and `y` to be constant values. The `in_axes` argument of `vmap` is used to specify over which argument we iterate in the loop. Here, we want to iterate over rng_key, but not on `x` and `y` and thus `in_axes` will be `(0,None,None)`. The `out_axes` argument is used to specify the output of the function. Use `out_axes=0` to get a 1D array of the results. The code looks like this:

```
bootstrap_jax = jax.vmap(one_bootstrap, in_axes=(0, None, None), out_axes=0)
```

To use it, we need to create the RNG keys and run the algorithm, and then create the `jnp.array` of the results. The code looks like this:

```
# key = jax.random.key(1234) # has already been done above
key, subkey = jax.random.split(key)
keys = jax.random.split(subkey, B)
beta_boot_jax = bootstrap_jax(keys, x1, y)
beta_boot_jax = jnp.vstack(beta_boot_jax)
se_boot_jax = jnp.std(beta_boot_jax, axis=0)
print(round(se_boot_jax, 3))
```

Another way to implement the loop in JAX is to use the `fori_loop` function in the `lax` module. The `fori_loop` function is a bit like the `reduce` function in Python. It iterates a function over a range of integers. The `state` variable contains all variables used and modified in the loop.

```
def body(i, state):
    key, beta_boot = state
    key, subkey = jax.random.split(key)
    # JAX version of beta_boot[i] = MLE(...):
    beta_boot = beta_boot.at[i].set(MLE(jnp.zeros(x1.shape[1]), 200, 0.005, x1, y))
    return key, beta_boot

key, subkey = jax.random.split(key)
state = (subkey, jnp.zeros((B, x1.shape[1])))
beta_boot_jax_v2 = jax.lax.fori_loop(0, B, body, state)[1]
```

Note that we have use `x = x.at[i].set(value)` instead of `x[i]=value` to update the array `beta_boot` in the `body` function since the latter is not allowed in JAX.

Part 2. The Bayesian logistic regression

At $\beta = (0, \dots, 0)$ the logistic model says that, whatever X_i, $[Y_i|X_i] \sim \text{Bernoulli}(0.5)$. Moreover, covariates have been standardized: they are comparable in the sense that they are in the same unit. Even the column that is constant, equal to 1 can be considered as a covariate of the same unit. Hence, without looking at the actual data, it is reasonable to assume that the prior distribution of β is centered at 0 and has variance $2 \times I_5$, where I_5 is the identity matrix:

$$\beta \sim \mathcal{N}(0, 2I_5).$$

This prior is a brake to large values of β that may appear when the parameters are overfitted to the data.

Now, we want to perform a Bayesian analysis of the data.

2.1 With PyMC

`PyMC` is a bit as if we replace the pain of probability calculus with the pain of installing and using a complex Python package. But it is worth it. Please, install it properly following the instructions at the beginning of the notebook.

The following lines of code define the Bayesian logistic regression model with PyMC. The prior distribution of β is a normal distribution with mean 0 and standard deviation 2 for each coordinate. The likelihood is a Bernoulli distribution with parameter $p(X_i)$ as defined above.

```
import pymc as pm
import arviz as az
x1np = np.array(x1)
ynp = np.array(y)
with pm.Model() as logistic_model:
    X = pm.Data('X', x1np)
    y_train = pm.Data('y', ynp)
    # prior
    beta = pm.Normal('beta', mu=0, sigma=np.sqrt(2), shape=5)
    # likelihood
    mu = pm.math.dot(X, beta)
    p = pm.Deterministic('p', pm.math.invlogit(mu))
    yobs = pm.Bernoulli('y_obs', p=p, observed=y_train)
pm.model_to_graphviz(logistic_model)
```

This is the DAG of the Bayesian model, a graphical representation of the dependencies between the variables. Nodes are in gray when observed, and in white when unobserved. The arrows indicate the dependencies between the variables that has been described in the code. The dimensions of the variables are indicated in the nodes. Check them to be sure that the model is correctly defined.

Next, we can run the sample to get a sample from the posterior distribution of β. We use 4 chains of 1000 iterations each, with 1000 tuning iterations first. The `cores` argument is set to 1 to avoid a bug with the `pymc` package and to use only one core of the CPU. In the `trace` object, we expect thus $4 \times 1000 = 4000$ draws from the posterior distribution.

```
with logistic_model:
    trace = pm.sample(1000, tune=1000, model=logistic_model, cores = 1,
    chains = 4)
```

We can use the `az.summary` function to get a summary of the posterior distribution of β.

```
az.summary(trace, var_names=["beta"])
```

A few plots:

```
az.plot_trace(trace, var_names="beta", compact=False)
az.plot_posterior(trace, var_names="beta", kind='kde')
az.plot_forest(trace, var_names="beta")
```

- The first series of plots shows that the four chains are sampled the same part of the parameter space. This is the expected result.
- The second series of plots shows the posterior distribution of each coordinate of β, with the kernel density estimate, and the 94% highest posterior density interval, which are credible intervals of probability 94%, and the mean of the posterior distribution.
- The final plot compares the credible intervals of probability 94% we get from the 4 chains. They should be equal, up to a Monte Carlo noise.

2.2 With a Metropolis-Hastings algorithm

We can also implement a Metropolis-Hastings algorithm to sample from the posterior distribution of β.

The first ingredient is a function that computes the log-posterior of β. Based on the log-likelihood function defined above, implement a log_posterior function that takes as input:

- `beta`: the parameter vector of the logistic regression model,
- `x`: the covariate matrix (including a first column of ones for the intercept),
- `y`: the response vector

and returns the log-posterior of β, using JAX and the @jit decorator.

```
# answer here
```

The second ingredient is the proposal kernel. In this simple situation we can propose a new value ζ from $\mathcal{N}(\beta, \Sigma)$. This is a symmetric kernel, so the log-acceptance ratio is the difference between the log-posterior at ζ and the log-posterior at β. Implement the Metropolis-Hastings algorithm as a function with the following input:

- `x`: the covariate matrix (including a first column of ones for the intercept),
- `y`: the response vector,
- `n_iter`: the number of iterations of the algorithm,
- `beta0`: the initial value of the parameter vector,
- `Sigma`: the covariance matrix of the proposal kernel.

The function should return the sample of the parameter vector and the acceptance rate.

```
# answer here
```

Run the algorithm on `x1` and `y` defined above with `n_iter=4500`, and `Sigma=0.002*np.eye(x1.shape[1])`, starting from the MLE, . Discard the first 500 draws as a burn-in period.

```
# answer here
```

Compare your results with the one from PyMC above.

Draw the five paths of the Metropolis-Hastings algorithm for each coordinate of β, starting at $t = 0$, as well as the resulting distribution of the last 4000 draws.

```
# answer here
```

Get a second chain by running the algorithm with the same parameters, but starting from $\beta = (0, \dots, 0)$.

```
# answer here
```

2.3 A Metropolis-Hastings algorithm with JAX

There is two major difficulties with JAX:

- using a random number generator (RNG) in a JAX function is not straightforward; (we have seen this problem in the previous section),
- iterative algorithms (where the state at time $t+1$ depends on the state at time t) are not straightforward either.

The second difficulty is the loop over the iterations. The main ingredient is to program on iteration of the Metropolis-Hastings algorithm as a function that takes as input:

- `rng_key`: the RNG key,
- `logpdf`: a function that computes the log-posterior of β,
- `logpdf_args`: the arguments of the log-posterior function,
- `position`: the current state,
- `log_prob`: the current log-probability of the current state.

The ouput should be the new state and the new log-probability.

```
from functools import partial

@partial(jax.jit, static_argnums=(1,))
def rw_metropolis_kernel(rng_key, logpdf, logpdf_args, position, log_prob):
    key, subkey = jax.random.split(rng_key)
    move = jax.random.normal(subkey, position.shape) * 0.045
    proposal = position + move
    log_prob_proposal = logpdf(proposal, *logpdf_args)
    log_uniform = jnp.log(jax.random.uniform(subkey))
    do_accept = log_uniform < log_prob_proposal - log_prob
    position = jnp.where(do_accept, proposal, position)
    log_prob = jnp.where(do_accept, log_prob_proposal, log_prob)
    return position, log_prob
```

Then, we can run the Metropolis-Hastings algorithm as follows. Along time, the position is stored into the `chain` object. The for-loop is replaced by a `jax.lax.fori_loop` function. The function that is iterated must have two inputs: the loop index `i` and the state of the loop, and must return the new state of the loop.

```
@partial(jax.jit, static_argnums=(1,2))
def rw_metropolis_sampler(rng_key, n_draws, logpdf, logpdf_args, initial_position):
    def mh_update(i, state):
        key, position, log_prob, chain = state
        _, key = jax.random.split(key)
        new_position, new_log_prob = rw_metropolis_kernel(key, logpdf, logpdf_args, position,
        chain = chain.at[i].set(new_position)
        return (key, new_position, new_log_prob, chain)
    logp = logpdf(initial_position, *logpdf_args)
    chain = jnp.zeros((n_draws, initial_position.shape[0]))
    rng_key, position, log_prob, chain = jax.lax.fori_loop(0, n_draws, mh_update, (rng_key, ini
    return chain
```

And to run 4 chains in parallel, we use the `vmap` function. We have to loop over the chains: the `in_axes` argument of `vmap` is used to specify over which argument we iterate in the loop. The `out_axes` argument is used to specify how the results are collected.

- The number of draws or iterations, the log-posterior function, and its arguments are the same for all chains. Thus two `0` are set in the `in_axes` argument, according to the position of these arguments in the function signature.
- The RNG key, the initial position, and the output chain are different for each chain. Thus, three `None`'s are set in the `in_axes` argument, according to the position of these arguments in the function signature.

The output chain is a 3D array, with the first dimension corresponding to the chains.

```
n_draws = 4500
n_chains = 4
key = jax.random.key(1234)
run_keys = jax.random.split(key, n_chains)
initial_positions = jnp.zeros((n_chains, x1.shape[1]))
run_mcmc = jax.vmap(rw_metropolis_sampler, in_axes=(0, None, None, None, 0), out_axes=0)
positions = run_mcmc(run_keys, n_draws, log_likelihood, (x1, y), initial_positions)
positions.block_until_ready()
print(positions.shape)
```

The output can be easily transformed into an arviz InferenceData object, and the results can be studied with the `az` functions.

```
trace2 = az.convert_to_inference_data({"beta": positions[:, 500:, :]})
print(az.summary(trace2, var_names=["beta"]))
az.plot_trace(trace2, var_names="beta", compact=False)
az.plot_posterior(trace2, var_names="beta", kind='kde')
az.plot_forest(trace2, var_names="beta")
```

It works! If your computer is fast enough, you can increase the number of draws to get a better approximation of the posterior distribution.

```
n_draws = 40500
n_chains = 4
key = jax.random.key(12345)
run_keys = jax.random.split(key, n_chains)
initial_positions = jnp.zeros((n_chains, x1.shape[1]))
run_mcmc = jax.vmap(rw_metropolis_sampler, in_axes=(0, None, None, None, 0), out_axes=0)
positions = run_mcmc(run_keys, n_draws, log_likelihood, (x1, y), initial_positions)
positions.block_until_ready()

trace3 = az.convert_to_inference_data({"beta": positions[:, 500:, :]})
print(az.summary(trace3, var_names=["beta"]))
az.plot_trace(trace3, var_names="beta", compact=False)
az.plot_posterior(trace3, var_names="beta", kind='kde')
az.plot_forest(trace3, var_names="beta")
```

2.4 Final remark

In section 2.2 and 2.3, we have used a proposal kernel that has been tuned to the posterior density. Choosing the correct variance matrix is crucial… and difficult. In practice, we use an adaptive MCMC methods, that tunes the variance matrix along the iterations of the burn-in period. This is a bit too complex for this first year course.