Empirical IO I: Problem Set 0 (Python Users)

Chris Conlon

Due: Sept 28, 2018

This problem set is designed to make sure that your `numpy` skills are up to speed. For each question, the expectation is that you complete the task, provide the appropriate code, and fully read the documentation, and work the `numpy` provided examples on your own. The tasks should be fairly easy. The goal is to accomplish the tasks in the most straightforward manner with the least amount of code.

If you are new to `numpy` I suggest the following tutorials:

- `https://docs.scipy.org/doc/numpy/user/quickstart.html`
- `https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html`

Part 0: Logit Inclusive Value

The logit inclusive value or $IV = \log \sum_{i=1}^{N} \exp[x_i]$.

1. Show that the this function is everywhere concave.
2. A common problem in practice is that if one of the $x_i > 600$ that we have an "overflow" error on a computer. In this case $\exp[600] \approx 10^{260}$ which is too large to store with any real precision, especially if another x has a different scale (say $x_2 = 10$). A common "trick" is to subtract off $m_i = \max_i x_i$ from all x_i. Show how to implement the trick and get the correct value of IV. If you get stuck take a look at Wikipedia.
3. Compare your function to `scipy.misc.logsumexp`. Does it appear to suffer from underflow/overflow? Does it use the max trick?

Part 1: Markov Chains

Consider the following Markov TPM:
Let $P = \{p_{i,j}\}$ be an $n \times n$ transition matrix of a Markov process where $\{p_{i,j}\}$ is interpreted as the probability that the system, when it is in state i, will move to state j. If we denote by $\pi_t[\pi_{t,1}, \pi_{t,2}, \ldots, \pi_{t,n}]$ the probability mass function of the system over the n states then $\pi_{t,j}$ evolves according to

$$\pi_{t+1,j} = \sum_{i=1}^{n} p_{i,j} \pi_{t,i}$$

Then we can write the state to state transition matrix as :

$$\begin{aligned} \pi_{t+1} &= \pi_t P \\ P &= \begin{bmatrix} 0.2 & 0.4 & 0.4 \\ 0.1 & 0.3 & 0.6 \\ 0.5 & 0.1 & 0.4 \end{bmatrix} \end{aligned}$$

We're interested in the ergodic distribution $\pi P = \pi$. This is similar to the transition matrix infinitely many periods into the future P^{∞}. Write a function that computes the ergodic distribution of the matrix P by examining the properly rescaled eigenvectors and compare your result to P^{100}. Here I recommend `numpy.linalg.matrix_power` and `numpy.linalg.eig`.

Part 2: Numerical Integration

In this part we will look to calculation the logit choice probability $p(X, \theta)$ by numerical integration:
$p(X, \theta) = \int_{-\infty}^{\infty} \frac{\exp(\beta_i X)}{1+\exp(\beta_i X)} f(\beta_i|\theta) \partial\beta_i$.
Assume $f \sim N(0.5, 2)$ and that $X = 0.5$.

1. Create the function in an Python called `binomiallogit`. (It should take β the item you integrate over as its argument, it should take the PDF `scipy.stats.norm.pdf` as an optional argument).
2. Integrate the function using Python's `scipy.integrate.quad` command and setting the tolerance to 1×10^{-14}. Treat this a the "true" value.
3. Integrate the function by taking 20 and 400 Monte Carlo draws from f and computing the sample mean.
4. Integrate the function using Gauss-Hermite quadrature for $k = 4, 12$ (Try some odd ones too). Obtain the quadrature points and nodes from the internet. Gauss-Hermite quadrature assumes a weighting function of $\exp[-x^2]$, you will need a change of variables to integrate over a normal density.[Wikipedia] You also need to pay attention to the constant of integration.
5. Compare results to the Monte Carlo results. *Make sure your quadrature weights sum to 1!*
6. Repeat the exercise in two dimensions where $\mu = (0.5, 1), \sigma = (2, 1)$, and $X = (0.5, 1)$.
7. Put everything into two tables (one for the 1-D integral, one for the 2-D integral). Showing the error from the "true" value and the number of points used in the evaluation.
8. Now Construct a new function `binomiallogitmixture` that takes a vector for X and returns a vector of binomial probabilities (appropriately integrated over $f(\beta_i|\theta)$ for the 1-D mixture). It should be obvious that Gauss-Hermite is the most efficient way to do this. *Do NOT use loops*

Part 3: Functional Approximation

What is interpolation? Suppose you have a collection of n points in R^2 and $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ where $y_i = g(x_i)$, $x_i \neq x_j$ $\forall i \neq j$ and you want to construct a function that closely fits these data points. Interpolation offers a variety of different techniques that perform this task in a more or less smooth manner.

We are interested in interpolating your `binomiallogitmixture`(x) from the previous part at 20 evenly spaced data points (`linspace`) from $[-4, 4]$. (You may want to work with an easier function such as `sin(x)` first.

We are going to use Chebyshev regression to construct a degree n polynomial that approximates a function f for $x \in [a, b]$ using $m > n$ points. The implementation requires the following steps, which you should include in your function `chebfit(fname, a, b, n)`.

1. Compute the m Chebyshev interpolation nodes on $[-1, 1]$, $z_k = -\cos\left(\frac{2k-1}{2m}\pi\right)$ for $k = 1, \ldots, m$.
2. Adjust the nodes to the $[a, b]$ interval $x_k = (1 + z_k)\left(\frac{b-a}{2}\right) + a$, for $k = 1, \ldots, m$.

3. Evaluate the function f at the approximation nodes $y_k = f(x_k)$ for $k = 1, \ldots, m$

4. Compute the Chebyshev polynomial coefficients via $a_i = \frac{\sum_{k=1}^{m} y_k T_i(z_k)}{\sum_{k=1}^{m} T_i(z_k)^2}$ where the polynomials $T_0(x) = 1, T_1(x) = x, T_{i+1}(x) = 2xT_i(x) - T_{i-1}(x)$.

5. Your function should return the nodes z_k and the coefficients c.

6. Write a new function that takes the coefficients c and computes a (vectorized) approximation to your function

$$\hat{f}(x) = \sum_{i=0}^{n} a_i T_i \left(2\frac{x-a}{b-a} - 1 \right)$$

7. Compare your approximation to Python's built-in chebyshev polynomial fitting `numpy.polynomial.chebyshev.chebfit` as well as the built in cubic-spline interpolator `scipy.interpolate`) and see how it does. (Plot both approximations on 0.01 spaced grid-points against the actual nodes).

8. What happens to the approximations outside the domain? What happens as we vary n?