QuantEcon Final Project

Emile DJOHI[*], Stephen EZIN[†], Adbel TIDJANI[‡]

1 Dynamic equations

- Write a Python function to simulate an AR(p) process
- Plot the simulated data.

Hint: - Here is the formula of an AR(p) process:

$$X_t = \sum_{i=1}^{t} \phi_i X_{t-i} + \epsilon_t$$

. - You can also generate ARMA processes using the `ArmaProcess` class from `statsmodels.tsa.arima_process`

2 Law of Large number and Central Limit theorem

- Generate a random sample of size 500 from a population with a finite mean and variance.
- Calculate the sample mean and sample standard deviation.
- Repeat steps 1 and 2 for a large number of times, say 1000 times.
- Plot a histogram of the sample means. How do we call the distribution of the sample means? What theoritical distribution does it follow? What is the name of this law?
- Calculate the mean and standard deviation of the sample means.
- Compare the mean and standard deviation of the sample means with the population mean and standard deviation.
- How do we call the standard deviation of the sample means. How to get the standard deviation of the population from the standard deviation of the samples means.
- What do you notice the more you increase the sample size.
- What is the name of this law?

[*]edjohi@africanschoolofeconomics.com
[†]sezin@africanschoolofeconomics.com
[‡]atidjani@africanschoolofeconomics.com

3 Monte-Carlo and option pricing

The least squares Monte Carlo (LSMC) method is a simulation method that combines a Monte Carlo simulation with a least-squares regression. It is a popular approach for pricing American options using Monte Carlo. You'll create a python class that represent an American put option. The class will have the necessary methods to implement LSMC in Python.

Here is a step-by-step algorithm to achieve this in python.

1. Import the necessary libraries:
    1. numpy: `import numpy as np`
    2. scipy.stats.norm: `from scipy.stats import norm`
    3. scipy.optimize.curve_fit: `from scipy.optimize import curve_fit`
2. Define the parameters of the option:
    1. Strike price (K)
    2. Time to maturity (T)
    3. Risk-free interest rate (r)
    4. Volatility (sigma)
    5. Number of time steps (N)
    6. Number of simulations (M)
3. Define the basis functions:
    1. The basis functions are defined in the `basis_functions` method of the AmericanPut class.
    2. The method takes in an array of values (x) and returns a matrix of basis functions.
    3. The basis functions used in this algorithm are: 1, x, x^2, x^3.
    4. You can use this function:

```
def basis_functions(self, x):
 return np.column_stack((np.ones_like(x), x, x**2, x**3))
```

4. Define the payoff function:
    1. The payoff function is defined in the `payoff` method of the AmericanPut class.
    2. The method takes in an array of stock prices (S) and returns an array of payoffs.
    3. The payoff for a put option is the maximum of the strike price minus the stock price and zero.
5. Generate the stock price paths:
    1. The `generate_paths` method of the AmericanPut class generates M simulations of N time steps of stock prices.
    2. The method uses the Black-Scholes model to simulate the stock prices.
    3. The method returns an M x (N+1) matrix of stock prices.
    4. In particular you can follow the following algorithm:
        - Calculate the time step size (dt) by dividing the time to maturity (T) by the number of time steps (N).
        - Set the initial stock price (S0) to 100.
        - Create an empty matrix (S) of size M x (N+1) to store the stock price paths. M represents the number of simulations, and (N+1) represents the number of time steps plus the initial time step.
        - Set the initial stock price for all simulations to S0 by assigning S0 to the first column of S.
        - Iterate over each time step from 0 to N-1:
            - Generate a random sample of size M from a standard normal distribution using

`np.random.normal(size=self.M)`. This random sample represents the random shocks to the stock price.
  - Calculate the stock prices at the next time step (i+1) for all simulations using the Black-Scholes model:
  - Multiply the current stock prices (S[:,i]) by the exponential of the drift term and the volatility term.
  - The drift term is `(self.r - 0.5*self.sigma**2)*dt`.
  - The volatility term is `self.sigma*np.sqrt(dt)*eps`, where `eps` is the random sample generated in the previous step.
  - Assign the calculated stock prices to the next column (i+1) of S.
- Return the matrix S, which contains the generated stock price paths.

6. Caculate the payoff for each stock price path:
   1. The `calculate_payoff` method of the AmericanPut class takes in the matrix of stock prices and returns the matrix of payoffs.
7. Perform backward induction:
   1. The `backward_induction` method of the AmericanPut class performs backward induction to calculate the option value.
   2. The method takes in the matrix of stock prices and payoffs and returns the matrix of option values.
   3. The method uses the least squares Monte Carlo (LSM) method to estimate the continuation value and exercise value at each time step.
   4. The method starts at the last time step and works backwards to the first time step.
   5. You can use the following algorithm:
8. Calculate the option value:
   1. The `calculate_option_value` method of the AmericanPut class takes in the matrix of option values and returns the average discounted option value.
   2. The method uses the risk-free interest rate to discount the option values.
9. Print the option value:
   1. The `calculate_option` method of the AmericanPut class calls the other methods in the correct order and prints the option value.
10. Instanciate the class:
    - Create an object of the class American named `opt1` using `opt1 = AmericanPut(K, T, r, sigma, N, M)`
    - Make sure to create the variables with the following values:

```
K = 100
T = 1
r = 0.05
sigma = 0.2
N = 100
M = 10000
```

11. Run the code:
    - Call the `calculate_option` method of the AmericanPut class to calculate the option value.
    - `option_value = option_pricing.calculate_option()`

4 Simple Bank Account System

You are tasked with creating a simple bank account system using Python classes. The system should be able to handle accounts, deposits, withdrawals, and transfers. Implement the following functionalities:

Create an Account class with the following attributes:

- Account number
- Account holder name
- Account balance

Implement methods for the Account class:

- Display account details
- Deposit money
- Withdraw money
- Transfer money to another account

Create a Bank class that keeps track of accounts. It should have methods to:

- Add an account to the bank
- Remove an account from the bank
- Display list of accounts in the bank

Test your Bank Account System by creating instances of the classes and performing various operations, such as creating accounts, making deposits and withdrawals, transferring money between accounts, and displaying information.

5 Bayesian updating

You are an economist studying the probability of recessions in Benin. You have historical data on the occurrence of recessions over several years. Your goal is to estimate the posterior distribution of the parameter theta, which represents the probability of a recession in a given year. The prior belief is a uniform distribution on $[0, 1]$.

To estimate the posterior distribution, you will use Bayesian updating. Bayesian updating is a method that allows you to update your beliefs about a parameter based on new data. It combines prior knowledge (prior belief) with observed data to obtain a posterior distribution that represents the updated belief.

In this case, you will update your belief about the probability of a recession in a given year based on the historical data on recessions. By using Bayesian updating, you can incorporate the prior belief and update it with the observed data to obtain the posterior distribution of theta.

The function you need to write will take a sequence of observations of recessions over time and update the belief after each observation. It will return the posterior distribution of theta on a grid of points between 0 and 1. The true value of theta is set to 0.314, and the number of observations is 100.

By using Bayesian updating, you can track the changes in the posterior distribution of theta as more observations are made. This allows you to continuously update your belief about the probability of a recession in Benin based on the available data.

1. Define the `prior distribution` of theta as a uniform distribution on $[0, 1]$.
2. For each data point, calculate the `likelihood of the data given theta` using the appropriate distribution. (what distribution does the event "Undergoing a recession" follows?)
3. Multiply the `prior distribution` by the `likelihood` to obtain the unnormalized posterior distribution.
4. Normalize the posterior distribution by dividing by its sum.
5. Repeat steps 2-4 for each data point to obtain the posterior distribution after each update.
6. To test your function, use the appropriate distibution you found to generate 100 random variables with $\theta = 0.314$. Feed it to your function. The peaks should be centered around 0.314

Here are some additional details about the code:

- You can start your function by creating a grid of points (`x_grid`) (100 points for instance) linearly spaced (`np.linspace()`)
- The `prior distribution` of theta is defined as a uniform distribution, meaning every point of your grid have the same probability. So, the `prior distribution` is given by an array of $\frac{1}{n}$, with n the number of points in your grid. Note that you can create an array of ones of size m using `np.ones(m)`
- The `likelihood` of the data given theta is calculated using the appropirate distribution. The approriate distribution is availble in the `scipy.stats` library. Let's call it `approp` for now. note that this is not the name of the approriate dist, you must look into the package and find the distribution by yourself. You can access the class, attributes, and methods of a package using `dir(package_name)`.
- Now you can compute the `likelihood` using `approp.pmf(v, 1, x_grid)`, where `v` is the data point and `x_grid` is an array of values for theta.
- The posterior distribution is obtained by multiplying the `prior distribution` by the `likelihood` and normalizing. This is implemented using `posterior = prior * likelihood` and `posterior = posterior / np.sum(posterior)`.
- The posterior distribution is visualized **after each update** using a kernel density estimate (KDE) plot. This is implemented using `sns.kdeplot(np.random.choice(x_grid, size=10000, p=posterior), color='black', alpha=0.1)`.