finm320_24_hw1

April 5, 2024

[1]:
```python
import numpy as np
from scipy.stats import norm
from scipy.optimize import bisect, brentq
from copy import copy
```

```
/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:146: UserWarning: A
NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy
(detected version 1.24.4
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

1 Problem 1

[2]:
```python
class UpAndOutPut:

    def __init__(self, K, T, barrier, observationinterval):
        self.K = K
        self.T = T
        self.barrier = barrier
        self.observationinterval = observationinterval
```

[3]:
```python
hw1contract = UpAndOutPut(K=95, T=0.25, barrier=114, observationinterval=0.02)
```

[4]:
```python
class GBMdynamics:

    def __init__(self, S, r, rGrow, sigma=None):
        self.S = S
        self.r = r
        self.rGrow = rGrow
        self.sigma = sigma

    def update_sigma(self, sigma):
        self.sigma = sigma
        return self
```

[5]:
```python
hw1dynamics = GBMdynamics(S=100, sigma=0.4, rGrow=0, r=0)
```

[6]:

```python
class TreeEngine:

    def __init__(self, N):
        self.N = N

    def price_upandout(self, dynamics, contract):

        deltat = contract.T / self.N
        # J is the level
        # np.ceil rounds up to integer: eg. 3.1 -> 4
        J = np.ceil(np.log(contract.barrier/dynamics.S)/(dynamics.sigma*np.sqrt(3*deltat))-0.5)
        # ensure price is checked at barrier level at right place
        deltax = np.log(contract.barrier/dynamics.S)/(J+0.5)

        # Possible stock price at maturity date T
        Sgrid = dynamics.S*np.exp(np.linspace(self.N, -self.N, num=2*self.N+1, endpoint=True)*deltax)
        #Here I decided to make the SMALLER indexes in this array correspond to HIGHER S

        numTimestepsPerObs = contract.observationinterval/deltat
        if abs(numTimestepsPerObs-round(numTimestepsPerObs)) > 1e-8:
            raise ValueError("This value of N fails to place the observation dates in the tree.")

        nu = dynamics.rGrow - dynamics.sigma**2 / 2
        Pu = 0.5 * ((dynamics.sigma**2 * deltat+ nu**2 * deltat**2)/deltax**2 + nu * deltat/ deltax)
        Pd = 0.5 * ((dynamics.sigma**2 * deltat+ nu**2 * deltat**2)/deltax**2 - nu * deltat/ deltax)
        Pm = 1 - (dynamics.sigma**2 * deltat + nu**2 * deltat**2)/deltax**2


        optionprice = np.maximum(contract.K-Sgrid,0)    #an array of time-T option prices.

        #Next, induct backwards to time 0, updating the optionprice array
        #Hint: if x is an array, then what are x[2:] and x[1:-1] and x[:-2]

        for t in np.linspace(self.N-1, 0, num=self.N, endpoint=True)*deltat:
            Sgrid = Sgrid[1:-1]
            optionprice_new = optionprice[1:-1].copy()
            optionprice_new = np.exp(-dynamics.r * deltat) * (optionprice[:-2] * Pu + optionprice[1:-1] * Pm + optionprice[2:] * Pd)
            if abs(t % contract.observationinterval) < 1e-8:
```

```
                optionprice_new[Sgrid >= contract.barrier] = 0
            optionprice = optionprice_new


        return optionprice[0]
        #The [0] is assuming that we are shrinking the optionprice array in␣
↪each iteration of the loop,
        #until finally there is only 1 element in the array.
        #If instead you are keeping unchanged the size of the optionprice array␣
↪in each iteration,
        #then you need to change the [0] to a different index.
```

1.0.1 Question a

[7]:
```
hw1tree=TreeEngine(N=100000)

up_and_out_put = hw1tree.price_upandout(hw1dynamics, hw1contract)
up_and_out_put
```

[7]: 5.301058572352042

1.0.2 Question b

[8]:
```
# Vanilla put = up-and-out put + up-and-in put
# up-and-in put = Vanilla put - up-and-out put
vanilla_contract = UpAndOutPut(K=95, T=0.25, barrier=100000000,␣
↪observationinterval=0.02)
vanilla_put = hw1tree.price_upandout(hw1dynamics, vanilla_contract)
up_and_in_put = vanilla_put - up_and_out_put
up_and_in_put
```

[8]: 0.2184798593080437

1.0.3 Question c1

The time-0 price of a continuously-monitored barrier option is generally smaller than the time-0 price of a discretely-monitored option. The continuous monitoring increases the likelihood of the barrier being breached (especially for knock-out options), thereby reducing the option's value due to the higher risk of the option becoming worthless before maturity. The discrete monitaring dates is a subset of the continuous monitaring dates, so the paths whose values are zero in the discrete case is a subset of the paths with value zero in the continuous case.

1.0.4 Question c2

Scenario 1: Here, the stock price S never reaches the barrier level before the expiry time T. For a non-knockout up-and-out put option, the outcome is identical to that of a standard vanilla put option, provided $S_T \leq 114$. This means, for the call option, since S_T cannot exceed 114 without triggering the barrier, the call option's value, represented as $(S_T - 136.8)^+$, equals 0. Consequently, the value from the vanilla put option in the portfolio is $(95-S_T)^+$, effectively replicating the barrier option's payoff in this scenario.

Scenario 2: When S hits or exceeds the barrier prior to T, the payoff defaults to zero, regardless of the specific time the barrier is breached. In this case, the goal is to adjust α so that the portfolio's value, composed of the vanilla put and $-\alpha$ units of the vanilla call, equals zero at any given time within $[0,T]$.

To solve for α, the strategy is to equalize the portfolio's value to zero at $t=0$, using the barrier level as the asset price. This establishes α as the ratio of the put option's value P_t to the call option's value C_t, i.e., $\alpha = \frac{P_t}{C_t}$.

By calculating this ratio at $t=0$ and assuming a zero interest rate ($r=0$), we find α analytically as:

$$\alpha = \frac{P_0}{C_0} = \frac{95\cdot N(-\frac{\log(114/95)}{0.4\cdot\sqrt{0.25}} + \frac{0.4\cdot\sqrt{0.25}}{2}) - 114\cdot N(-\frac{\log(114/95)}{0.4\cdot\sqrt{0.25}} - \frac{0.4\cdot\sqrt{0.25}}{2})}{114\cdot N(\frac{\log(114/136.8)}{0.4\cdot\sqrt{0.25}} + \frac{0.4\cdot\sqrt{0.25}}{2}) - 136.8\cdot N(\frac{\log(114/136.8)}{0.4\cdot\sqrt{0.25}} - \frac{0.4\cdot\sqrt{0.25}}{2})} = \frac{5}{6} \approx 0.8333$$

[9]:
```
# P(St=114) - alpha * C(St=114) = 0
# alpha = P(St=114) / C(St=114)
def black_scholes_call(S, K, T, r, sigma):
    # Calculate d1 and d2
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    # Calculate the call price
    call_price = (S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2))
    return call_price

def black_scholes_put(S, K, T, r, sigma):
    # Calculate d1 and d2
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    # Calculate the put price
    put_price = (K * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1))
    return put_price

put_price = black_scholes_put(S=114, K=95, T=0.25, r=0, sigma=0.4)
call_price = black_scholes_call(S=114, K=136.8, T=0.25, r=0, sigma=0.4)
alpha = put_price/call_price
print("alpha is " + str(round(alpha,3)))

put_price_rep = black_scholes_put(S=100, K=95, T=0.25, r=0, sigma=0.4)
call_price_rep = black_scholes_call(S=100, K=136.8, T=0.25, r=0, sigma=0.4)
option_price = put_price_rep - alpha * call_price_rep
print("time-0 value of the continuously-monitored barrier option is " +
 ↪str(round(option_price,3)))
```

```
alpha is 0.833
```

time-0 value of the continuously-monitored barrier option is 5.032

2 Problem 2

[10]:
```python
# uses the same GBMdynamics class as in Problem 1
```

[11]:
```python
class CallOption:

    def __init__(self, K, T, price=None):
        self.K = K
        self.T = T
        self.price = price
```

[12]:
```python
class AnalyticEngine:

    def __init__(self):
        pass

    def BSpriceCall(self, dynamics, contract):
        # ignores contract.price if given, because this function calculates␣
 ↪price based on the dynamics

        F = dynamics.S*np.exp(dynamics.rGrow*contract.T)
        std = dynamics.sigma*np.sqrt(contract.T)
        d1 = np.log(F/contract.K)/std+std/2
        d2 = d1-std
        #return call option price using black scholes
        return np.exp(-dynamics.r*contract.T)*(F*norm.cdf(d1)-contract.K*norm.
 ↪cdf(d2))

    def IV(self, dynamics, contract):
        # ignores dynamics.sigma, because this function solves for sigma.

        if contract.price is None:
            raise ValueError('Contract price must be given')

        df = np.exp(-dynamics.r*contract.T)  #discount factor
        F = dynamics.S / df
        lowerbound = np.max([0,(F-contract.K)*df])
        C = contract.price
        if C<lowerbound:
            return np.nan
        if C==lowerbound:
            return 0
        if C>=F*df:
            return np.nan
```

```
        dytry = copy(dynamics)
        # We "try" values of sigma until we find sigma that generates price C

        # First find lower and upper bounds
        sigma_try = 0.2
        while self.BSpriceCall(dytry.update_sigma(sigma_try),contract)>C:
            sigma_try /= 2
        while self.BSpriceCall(dytry.update_sigma(sigma_try),contract)<C:
            sigma_try *= 2
        hi = sigma_try
        lo = hi/2
        # We have calculated "lo" and "hi" which bound the implied volatility from below and above.
        # In other words, the implied volatility is somewhere in the interval [lo,hi].
        # Then, to calculate the implied volatility within that interval,
        # for purposes of this homework, you may either (A) write your own bisection algorithm,
        # or (B) use scipy.optimize.bisect or (C) use scipy.optimize.brentq
        # You will need to provide lo and hi to those solvers.
        # There are other solvers that do not require you to bound the solution
        # from below and above (for instance, scipy.optimize.fsolve is a useful solver).
        # However, if you are able to bound the solution (of a single-variable problem),
        # then bisection or Brent will be more reliable.


        #returns the difference between the calculated Black-Scholes price and the market price for a given volatility
        def IV_root(sigma):
            return self.BSpriceCall(dytry.update_sigma(sigma),contract) - C

        impliedVolatility = bisect(IV_root, lo, hi)

        return impliedVolatility
```

[13]:
```
#Test the BSpriceCall function
hw1analytic = AnalyticEngine()
dynamics2 = GBMdynamics(sigma=0.4, rGrow=0, S=100, r=0)
contract2 = CallOption(K=100, T=0.5)
hw1analytic.BSpriceCall(dynamics2,contract2)
```

[13]: 11.246291601828489

[14]:
```
#Test the IV function
contract2.price = 12
hw1analytic.IV(dynamics2,contract2)    # This code, EXACTLY AS WRITTEN HERE,␣
↪must execute without crashing
```

[14]: 0.427005424113304

2.0.1 Question a

[15]:
```
contract2.price = 11.25
contract3 = CallOption(K=100, T=1, price=12)
IV1 = hw1analytic.IV(dynamics2,contract2)
IV2 = hw1analytic.IV(dynamics2,contract3)
print("The time-0 Black-Scholes implied volatilities for 0.5 year European call␣
↪is " + str(IV1))
print("The time-0 Black-Scholes implied volatilities for 1 year European call␣
↪is " + str(IV2))
```

```
The time-0 Black-Scholes implied volatilities for 0.5 year European call is
0.40013278092228577
The time-0 Black-Scholes implied volatilities for 1 year European call is
0.3019384309925955
```

2.0.2 Question b

[16]:
```
# Part (b)
contract4 = CallOption(K=100, T=0.75)
dynamics3 = GBMdynamics(sigma= (IV1 + IV2)/2, rGrow=0, S=100, r=0)
price_75 = round(hw1analytic.BSpriceCall(dynamics3,contract4), 3)
print("The time 0 price of the 0.75-expiry call is " + str(price_75))
```

```
The time 0 price of the 0.75-expiry call is 12.082
```

2.0.3 Question c

Assume cash settlement.

At time 0, short 1 unit of the 0.75-expiry call option in part (b), and long 1 unit of the 1-year expiry call option in part (a). The net value at time 0 is calculated as 12 - 12.082 = -0.082.

When $S_{0.75} <= 100$, the 0.75-year expiry call option expires worthless, but the 1-year expiry call option still has value which result in a zero or positive net value at time 1. This is Type-2 arbitrage

When $S_{0.75} > 100$, the 0.75-year expiry call option is executed. Thus, we sell 1-year expiry call option to pay the money $S_{0.75} - 100$. The net value = price of 1-year expiry call at $0.75 - (S_{0.75} - 100) >= (S_0.75 - 100) - (S_0.75 - 100) = 0$ because price of a call option is always at least as great as its intrinsic value. Therefore, the net value at time 0.75 is non-negative, leading to Type-2 arbitrage

[ ]: