HW7

May 17, 2024

1 Question 1

1.1 (a)

Two of the functions have one or more places where you are to “FILL THIS IN”

[1]:
```
import numpy as np
import scipy.optimize
import statsmodels.api as sm
```

```
/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}"
```

[2]:
```
class GBM:

    def __init__(self,S0,r,sigma):
        self.S0 = 1
        self.r = 0.03
        self.sigma = 0.20
```

[3]:
```
hw7dynamics = GBM(S0=1,r=0.03,sigma=0.20)
```

[4]:
```
class Put:

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

[5]:
```
hw7contract = Put(K=1.1,T=4)
```

[8]:
```
class MCengine:

    def __init__(self,M,N,seed,algorithm):

        self.M = M # Number of paths
        self.N = N     # Number of time periods
```

```
        self.rng = np.random.default_rng(seed=seed) # Seeding the random number␣
↪generator with a specified number helps make the calculations reproducible

        self.algorithm = algorithm
        #'value' for Value-based approach (Longstaff-Schwartz) -- problem 1a
        #'policy' for Policy optimization -- problem 1b

    def price_americanPut_GBM(self,contract,dynamics):

        r=dynamics.r
        sigma=dynamics.sigma
        S0=dynamics.S0

        K=contract.K
        T=contract.T

        N=self.N
        M=self.M
        dt=T/N

        Z = self.rng.normal(size=(M,N))

        paths = S0*np.exp((r-sigma**2/2)*dt*np.tile(np.
↪arange(1,N+1),(M,1))+sigma*np.sqrt(dt)*np.cumsum(Z,axis=1))

        payoffDiscounted = np.maximum(0,K-paths[:,-1])

        for nn in np.arange(N-1,0,-1):
            continuationPayoffDiscounted = np.exp(-r*dt)*payoffDiscounted

            X=paths[:,nn-1]
            exerciseValue = K-X

            if self.algorithm == 'value':
                # This is the value function (Longstaff-Schwartz) approach. ␣
↪For problem 1a

                # should be 1, X, X^2 matrix
                # set axis = 1 so that the matrix will make each row a column
                basisfunctions = np.stack([np.ones(M), X, X**2], axis = 1)
                # FILL THIS IN.  You may use np.stack
                # This will be an M-by-3 array containing the basis functions␣
↪(Same ones as L7.9-7.10, and Excel)

                # find beta by OLS, y should eb continuationpayoffdiscounted, x␣
↪should be the basis function
```

```
                # note that we should use only in-the-money paths to fit the␣
↪regression model
                model_ols = sm.
↪OLS(continuationPayoffDiscounted[exerciseValue>0],␣
↪basisfunctions[exerciseValue>0])
                ols_results = model_ols.fit()
                coefficients = ols_results.params
                # FILL THIS IN
                # This will be an array of 3 estimated "betas".

                # predicted values
                # we can use basisfunctions @ coefficients or ols_results.
↪predict(basisfunctions)
                estimatedContinuationValue = basisfunctions @ coefficients
                # FILL THIS IN with an array of length M.
                # This is similar to the Red column in Excel

                whichPathsToExercise = (exerciseValue >= np.
↪maximum(estimatedContinuationValue,0))
                        #This is a length-M array of Booleans

            elif self.algorithm == 'policy':
                # This is the policy optimization approach to Reinforcement␣
↪learning.  For problem 1b

                (a_opt,b_opt) = scipy.optimize.minimize(
                   ␣
↪negofMCaverageOfExpectedPayouts,(0,0),args=(X,exerciseValue,continuationPayoffDiscounted),m
↪x
                    #Chose Nelder-Mead optimizer because it is generating␣
↪reasonable results with minimal coding effort
                    #But gradient methods, done properly, usually run faster

                # get the optinmized a and b above and get the optimized p
                p_opt = softExercise(X, a_opt, b_opt)
                # ensure the p should not be smaller than 0.5 and the exercise␣
↪value should not be OTM
                whichPathsToExercise = ((p_opt>=0.5) & (exerciseValue>0))
                    #FILL THIS IN, using the right-hand side of the last␣
↪equation on the homework sheet
                    #This obtains the hard exercise decision from the optimized␣
↪soft exercise function
                    #It should be a length-M array of Booleans (as it was in␣
↪the "value" approach.
                    #But here it comes from the softExercise function)
```

```
            else:
                raise ValueError('Unknown algorithm type')

            # whichpathstoexercise tells you if the exercise is larger than␣
↪expected continuation values
            # if true, you should exercise so take exercise value
            # if false, use the discounted continuation cash flow as the excel␣
↪shows
            payoffDiscounted[whichPathsToExercise] =␣
↪exerciseValue[whichPathsToExercise]# FILL THIS IN -- see the "discounted␣
↪cashflow along path" column in Excel
            payoffDiscounted[np.logical_not(whichPathsToExercise)] =␣
↪continuationPayoffDiscounted[np.logical_not(whichPathsToExercise)]# FILL␣
↪THIS IN -- see the "discounted cashflow along path" column in Excel

        # The time-0 calculation needs no regression
        continuationPayoffDiscounted = np.exp(-r*dt)*payoffDiscounted;
        estimatedContinuationValue = np.mean(continuationPayoffDiscounted);
        putprice = max(K-S0,estimatedContinuationValue);

        return(putprice)
```

[6]:
```
# for Policy optimization approach, problem 1b
#
# If b<<0 then this function essentially returns nearly 1 if X<a, or nearly 0␣
↪if X>a
# but with some smoothing of the discontinuity, using a sigmoid function, to␣
↪help the optimizer

def softExercise(X,a,b):
    return 1/(1+np.exp(-b*(X-a)))
```

[7]:
```
# for Policy optimization approach, problem 1b

def negofMCaverageOfExpectedPayouts(coefficients, x, exercisePayoff,␣
↪continuationPayoff):

    p = softExercise(x,*coefficients)

    # p and exercisePayoff and continuationPayoff are all length-M arrays

    # p is result of the softexercise results as showing above
    return -np.mean(p*exercisePayoff + (1-p)*continuationPayoff)

## You fill in, what to return.  It should be the negative of the expression␣
↪inside the max() on the homework sheet.
```

```
## Need to take the negative because we are calling "minimize" but we want to␣
↪do _maximization_
```

[9]:
```
hw7MC = MCengine(M=10000,N=4,seed=0,algorithm='value')
hw7MC.price_americanPut_GBM(hw7contract,hw7dynamics)
```

[9]: 0.16210625617317392

1.2 (b)

[10]:
```
hw7MC = MCengine(M=10000,N=4,seed=0,algorithm='policy')
hw7MC.price_americanPut_GBM(hw7contract,hw7dynamics)
```

```
/var/folders/x9/dklkzy9s3rj0cq0ryk2h2zcc0000gn/T/ipykernel_41604/3835755103.py:7
: RuntimeWarning: overflow encountered in exp
  return 1/(1+np.exp(-b*(X-a)))
```

[10]: 0.16263529459015832

2 Question 2

[11]:
```
import numpy as np
from scipy.stats import norm
from copy import copy
```

[12]:
```
class GBM:

    def __init__(self,sigma,r,drift,S_t,t):
        self.sigma = sigma
        self.r = r
        self.drift = drift    #sometimes we denoted this as "rGrow"
        self.S_t = S_t
        self.t = t
```

[13]:
```
class CallOption:

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

[14]:
```
class CallBinary:

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

[15]:
```
class EYcontract:

    def __init__(self,threshold0,K_case1,K_case2lower,K_case2upper,T0,T1):
        self.K_case1 = K_case1
        self.K_case2lower = K_case2lower
        self.K_case2upper = K_case2upper
        self.threshold0 = threshold0
        self.T0=T0
        self.T1=T1
```

[16]:
```
class AnalyticEngine:

    def __init__(self):
        pass

    def BSpriceCall(self, dynamics, contract):
        timeRemaining = contract.T - dynamics.t
        F = dynamics.S_t*np.exp(dynamics.drift*timeRemaining)
        std = dynamics.sigma*np.sqrt(timeRemaining)
        d1 = np.log(F/contract.K)/std+std/2
        d2 = d1-std
        return np.exp(-dynamics.r*timeRemaining)*(F*norm.cdf(d1)-contract.
↪K*norm.cdf(d2))

    def BSpriceCallBinary(self, dynamics, contract):

        #fill this in
        timeRemaining = contract.T - dynamics.t
        F = dynamics.S_t*np.exp(dynamics.drift*timeRemaining)
        std = dynamics.sigma*np.sqrt(timeRemaining)
        d1 = np.log(F/contract.K)/std+std/2
        d2 = d1-std

        return np.exp(-dynamics.r*timeRemaining)*norm.cdf(d2)
```

[17]:
```
class MCengine:

    def __init__(self, M, seed):
        self.M = M                                  # How many simulations
        self.rng = np.random.default_rng(seed=seed) # Seeding the random number␣
↪generator with a specified number helps make the calculations reproducible

    def price_EYcontract(self,contract,dynamics):

        timeUp = contract.T0 - dynamics.t
```

```
        S_T0 = dynamics.S_t * np.exp((dynamics.drift - (1/2)*dynamics.
↪sigma**2)*timeUp + dynamics.sigma* np.sqrt(timeUp)*self.rng.
↪standard_normal(self.M))

        fs_results = []

        for sim_s in S_T0:
            dynamicsConditional = copy(dynamics)
            # now we will consider T0 as time 0
            # the starting price will be simulated T0 stock price
            dynamicsConditional.t = contract.T0
            dynamicsConditional.S_t = sim_s

            if sim_s > contract.threshold0:
                embeddedCallcase1 = CallOption(K=contract.K_case1, T=contract.
↪T1)
                conditional_payoff = AnalyticEngine().
↪BSpriceCall(dynamicsConditional,embeddedCallcase1)
            else:
                embeddedCallcase2 = CallOption(K=contract.K_case2lower,␣
↪T=contract.T1)
                embeddedBinarycase2 = CallBinary(K=contract.K_case2upper,␣
↪T=contract.T1)
                # according to hint in class, it will be normal BS call -␣
↪binary call with upper K
                conditional_payoff = AnalyticEngine().
↪BSpriceCall(dynamicsConditional,embeddedCallcase2) - AnalyticEngine().
↪BSpriceCallBinary(dynamicsConditional, embeddedBinarycase2)

            fs_results.append(conditional_payoff)

        # get each simulated time 0 value
        all_fs = np.exp(-dynamics.r * timeUp) * np.array(fs_results)
        # take expected values as the average of above
        price = np.mean(all_fs)
        standard_error = np.std(all_fs, ddof=1)/np.sqrt(self.M)

        return(price, standard_error)
```

[18]:
```
hw7p2contract=EYcontract(threshold0=12,K_case1=11,K_case2lower=10,K_case2upper=14,T0=0.
↪5,T1=1.0)
```

[19]:
```
hw7p2dynamics=GBM(sigma=0.7,r=0.02,drift=0.02,S_t=10,t=0)
```

[20]:
```
hw7p2MC=MCengine(M=100000,seed=0)
(price, std_err) =  hw7p2MC.price_EYcontract(hw7p2contract,hw7p2dynamics)
print(price,std_err)
```

```
2.5452351389161993 0.011448028772557314
```

[ ]: