University of California, Santa Barbara

PSTAT 176/276

FINAL PROJECT

American Put Option Estimator

Authors:
Emily Lu
Liang Feng
Mayuresh Anand
Niall Lawler
Taiga Schwarz

Supervisor:
Professor Tao Chen

June 13th, 2020

Abstract

In this project, we are interested in estimating the American put option price of the stock, Alphabet Class C (ticker: GOOG). To do this, we utilized the programming language, Python, to first calibrate the stock volatility under the Geometric Brownian Motion model, second, apply the stochastic process of Random walk to stimulate the stock price paths, third, generate the European put option price through the Monte Carlo method, and fourth, apply a regression technique using machine learning to generate the American put option price. Afterwards, given our final American put option price, we applied a control variate method and analyzed whether or not our final American put option price would benefit from it. We found that the American put option price benefited from the control variate method.

Contents

1 Introduction

On March 9th, 12th, and 16th of the year 2020, the Dow Jones Industrial Average Index (DJIA) experienced one of its worst point drops in U.S. history. For many investors, these three days represented also the worst point drops in their portfolio value. However, investors holding put options may have fared otherwise. Put options are contracts that give holders the right, but not the obligation, to sell a specified amount of an underlying security at a pre-determined price. Thus, imagine if we had purchased put options to hedge our portfolios, then we perhaps could have perserved our portfolio returns during the market case. This brings us to our project objective. Suppose we forecasted this market crash and wanted to hedge our portfolio or to benefit from this downfall, then at what price should we buy our put option for? Or more specifically, at price would a market maker be willing to sell the put option to us such that he/she is protected from arbitrage - the simultaneous purchase and sale of an asset to profit from a premium imbalance?

Therefore, in this project, we will be applying modern computational finance methods to estimate the price of an American Put Option with a stock as the underlying asset. Our selected stock will be "Alphabet Class C" (ticker: GOOG) which is a holding company that engages in the business of acquisition and operations of other businesses such as Google. As of June 2020, GOOG has since recovered from the March 2020 crash; however, we believe that the market will be due for another correction so we are interested in purchasing a put option that is priced fairly. To determine the fair price option price, we will be utilizing the programming language, Python, to first calibrate the stock volatility under the Geometric Brownian Motion model, second, apply the stochastic process of Random walk to stimulate the stock price paths, third, generate the European put option price through the Monte Carlo method, and fourth, apply a regression technique using machine learning to generate the American put option price. Afterwards, given our final American put option price, we would like to apply a control variate method and analyze whether or not our final American put option price would benefit from it.

2 Stock Volatility Estimation

Before we begin estimating GOOG's price path, we need to determine the historical volatility - or the average deviation from the average price - of GOOG. To do so, we will model the stock price using the Geometric Brownian model. Given the historical prices: $S_0, S_h, S_{2h}, \ldots, S_{252h}$ prices for 252 trading days in a year, let's assume

$$S_{nh} = S_{(n-1)h} e^{\left(r - \frac{\sigma^2}{2}\right)h + \sigma Z_n \sqrt{h}}, \tag{1}$$

where S_{nh} is the stock price at time nh, r is the interest rate, σ is the stock volatility, h is the time step, and Z_n is a normally distributed random variable. Using the price model, we could mathematically solve for σ by doing the following derivations:

$$S_{nh} = S_{(n-1)h} e^{\left(r - \frac{\sigma^2}{2}\right)h + \sigma Z_n \sqrt{h}} \iff \frac{S_{nh}}{S_{(n-1)h}} = e^{\left(r - \frac{\sigma^2}{2}\right)h + \sigma Z_n \sqrt{h}} \tag{2}$$

$$\iff ln\left(\frac{S_{nh}}{S_{(n-1)h}}\right) = \left(r - \frac{\sigma^2}{2}\right)h + \sigma Z_n \sqrt{h}. \tag{3}$$

Afterwards, let $\hat{S_n} = ln\left(\frac{S_{nh}}{S_{(n-1)h}}\right)$ such that $\hat{S_n} \sim N\left(\left(r - \frac{\sigma^2}{2}\right)h, \sigma^2 h\right)$ and $Var(\hat{S}) = \sigma^2 h$ for $\hat{S_1}, \hat{S_2}, \ldots, \hat{S_{252}}$. Then, using our sample data, we get $\sigma = \sqrt{\frac{Var(\hat{S})}{h}}$.

We implemented this using the following python code below:

```
def StockVol(histoPrice):

    Sn = [np.log(histoPrice[i+1]/histoPrice[i]) for i in range(len(histoPrice)-1)]

    return np.sqrt(np.var(Sn, ddof=1))
```

3 Stock Price Path Simulation

After computing the historical volatility of GOOG, we could then simulate price paths. The simulation of stock prices should return a $m \times n$ matrix where m is the number of time periods and n is the number of stock paths. So given $[0, T]$ with n time steps, we have

$$S_0, S_{T/n}, S_{2T/n}, \ldots, S_{(n-1)T/n}, S_T,$$

where

$$S_{T/n} = S_0 e^{\left(r-\frac{\sigma^2}{2}\right)\frac{T}{n}+\sigma Z\sqrt{\frac{T}{n}}},$$

$$S_{2T/n} = S_{T/n} e^{\left(r-\frac{\sigma^2}{2}\right)\frac{T}{n}+\sigma Z\sqrt{\frac{T}{n}}}$$

and so on... Each row of the stock price could be denoted as $Y_i = e^{\left(r-\frac{\sigma^2}{2}\right)\frac{T}{n}+\sigma Z_i\sqrt{\frac{T}{n}}}$.

We implemented this using the following python code below:

```
def StockPath(nSteps, sigma, S0, T, nSimulations, r, delta):

    path = []
    for period in range(nSimulations):
        step =  T/nSteps
        periodPrice = [S0]
        Z = np.random.normal(0, 1, nSteps)
        Y = np.exp(((r-delta-(sigma**2)/2)*step)+(sigma*Z*np.sqrt(step)))
        counter = 0

        for y in Y:
            periodPrice.append(periodPrice[counter]*y)
            counter += 1
        path.append(periodPrice)

    return(path)
```

4 European Put Option Price through Monte Carlo Method

The European put option is a contract that could only be exercised at its terminal time. Thus, given this, we are interested in computing the European put option price of GOOG to compare it

to our American put option price. In general, the European put option price is modelled by

$$\begin{aligned} \text{Premium} &= e^{-rT}\mathbb{E}[(K - S_t)^+] \quad (4) \\ &\approx e^{-rT}\text{avg}[(K - S_t)^+]. \quad (5) \end{aligned}$$

To solve for this, we calculate for the average discounted returns over all paths.

The following Python code below implements the European put option computation:

```
def EurOptPrice(StockPath, T, r, K):

    # number of simulations
    n = np.shape(StockPaths)[0]
    # number of timesteps
    nsteps = np.shape(StockPaths)[1]
    # Last Column of StockPath is Terminal Value
    Dis_Payoff_Vec = []

    for j in range(0,n):
        ST_j = StockPath[j][nsteps-1]
        # Create Column of Disounted Payoffs
        Dis_Payoff_j = np.exp(-r*T)*max(0,K - ST_j)
        Dis_Payoff_Vec = np.append(Dis_Payoff_Vec,Dis_Payoff_j)
        Price = np.mean(Dis_Payoff_Vec)
        Price_Var = np.var(Dis_Payoff_Vec)

    return (Dis_Payoff_Vec, Price, Price_Var)
```

5 American Put Option Price without Control Variates

An American option could be exercised at any time during the contract, unlike a European option which could only be exercised at its terminal time. One should exercise an American option at the first time $t \in [0, T]$ when the discounted payoff of exercising at that time is greater than the expected payoff of continuing to hold onto the option. The payoffs at each time are easy to compute, however the discounted value of holding onto the option and also the value of the option at each time must be computed backwards starting from $t = T$. For an American put option, we could write this mathematically as solving for

$$V_t = max\{(K - S_t)^+,\ e^{-r\Delta t}\mathbb{E}_Q[V_{t+1}|S_t]\}$$

for each $t \in [0, T]$ moving backwards from $t = T$.

At $t = T$, the value of the option is simply equal to the payoff because it is the last chance to exercise. Mathematically, we could write this as

$$V_T = (K - S_T)^+.$$

For $t = T - 1, T - 2, ..., 0$, we need to solve

$$V_t = max\{(K - S_t)^+,\ e^{-r\Delta t}\mathbb{E}[V_{t+1}|S_t]\}.$$

We could calculate $(K - S_t)^+$ using our input value for K and S_t from **StockPaths**, our simulated stock paths using the Black-Scholes model. On the other hand, there is no closed-form solution to calculate $\mathbb{E}[V_{t+1}|S_t]$. Instead, we could estimate its value by doing an additional 1-step Monte Carlo at each time step along with a machine learning model. Let us illustrate the algorithm with an example:

Let $t = T - 1$. Suppose we have N number of simulated stock paths. At each S_{T-1}^j, where $j = 1, ..., N$ and denotes the stock path, we do a 1-step Monte Carlo simulation to get M number of S_T values. Then using the simulated S_T values, we could estimate $\mathbb{E}[V_T|S_{T-1}]$ for each path:

$$\mathbb{E}[V_T|S_{T-1}] \approx \frac{1}{M}\sum_{m=1}^{M}(K - S_T^m)^+, \quad \text{where m is the mth simulated } S_T.$$

We could then substitute what we compute for $\mathbb{E}[V_T|S_{T-1}]$ into

$$V_{T-1} = max\{(K - S_{T-1})^+,\ e^{-r\Delta t}\mathbb{E}[V_T|S_{T-1}]\}.$$

If we do this for every path in **StockPaths**, we get a training set for our machine learning model:

$$(S_{T-1}^1,\ V_{T-1}^1),\ (S_{T-1}^2, V_{T-1}^2),\ ...,\ (S_{T-1}^N, V_{T-1}^N).$$

To be specific, we use quadratic regression on the training set to give us a model of option value against stock price, i.e. $f(S_{T-1}) := V_{T-1}$. [For this we use **cvpxy** library which could automatically solve the problem and minimize the cost function to estimate the best parameter. A variable **beta** is defined of three elements corresponding to $(1, \beta, \beta^2)$. We then define the loss function using sum of squares on **discountedExpectedPayoffs** and **X** matrix which has entries $[1, \text{stockPrice}, \text{stockPrice}^2]$. We then formulate the problem on this loss function, which is then solved to estimate parameter β to minimize the loss function.]

Next, for $t = T - 2$, we want

$$V_{T-2} = max\{(K - S_{T-2})^+,\ e^{-r\Delta t}\mathbb{E}[V_{T-1}|S_{T-2}]\}.$$

Again we do 1-step Monte Carlo at each S_{T-2} in **StockPaths** in order to compute $\mathbb{E}[V_{T-1}|S_{T-2}]$. We need V_{T-1} values for every S_{T-1} we simulate in the 1-step Monte Carlo. We could use the regression model $f(S_{T-1})$ we constructed in the previous time step to get the V_{T-1} values:

$$V_{T-1}^m = f(S_{T-1}^m), \quad m = 1, ..., M.$$

Then, we could compute $\mathbb{E}[V_{T-1}|S_{T-2}]$ as

$$\mathbb{E}[V_{T-1}|S_{T-2}] = \frac{1}{M}\sum_{m=1}^{M} f(S_{T-1}^m).$$

Then we substitute this value into the equation for V_{T-2}. Now that we have V_{T-2} and S_{T-2} for each path, we could again create a regression model of V_{T-2} against S_{T-2}.

We continue this process moving backwards in time until we reach $t = 0$ and thus we would be able to solve for V_0, giving us the discounted payoff of the option. Taking the mean of the discounted option value V_0 of each path gives us an approximation of the option price. We implemented this using the following Python code below:

```
def AmericanOptionPrice(StockPaths, sigma, r, K, T, mcSamples, delta):

    nSimulations = np.shape(StockPaths)[0]
    nSteps = np.shape(StockPaths)[1] - 1
    step = T/nSteps
    paths = np.array(StockPaths).T[::-1]

    # This contains payoffs of all the path or 0th time step
    exercisedPayoffs = [(K-paths[0]).clip(min=0)]
    exercisedExpDisPayoffs = [(K-paths[0]).clip(min=0)]

    for i in range(1,nSteps):
        stockPrices = paths[i]
        payOffs = (K - stockPrices).clip(min=0)
        simulatedPrices = [stockPrice*np.exp(sigma*np.random.normal(0,1,mcSamples)
                                *np.sqrt(step) + (r-(sigma**2)/2)*step)
                                for stockPrice in stockPrices]
        simulatedPrices = np.array(simulatedPrices)

        #If this is the t-1 step then the expected payoff is to be simulated mean
        if(i == 1):
            expectedPayoffs = np.array([np.mean(expectedPayoff)
                            for expectedPayoff in K-simulatedPrices.clip(min=0)])
            discountedExpectedPayoffs = expectedPayoffs* np.exp(-r*step)

        # For other time step it must be done by machine learning
        else:
            """
            discountedExpectedPayoffs is calculated through machine learning
            Takes in stockPrices (has stock price for each path at time step i)
            and discountedExpectedPayoffs (has expectedPayoffs
            for each path at time step i) to run the machine learning
            model on this to get the weights to use this weights to
            predict new expected payoff at timestep i-1
            """

            X = np.array([np.array([1]*len(stockPrices)),
                                    stockPrices, stockPrices**2])
            X = X.T

            # Formalizing the regression model
            beta = cp.Variable(3)
            loss = cp.sum_squares(discountedExpectedPayoffs-X@beta)
            prob = cp.Problem(cp.Minimize(loss))
            prob.solve()
            # estimating discounted expected payoffs using estimated parameters
            discountedExpectedPayoffs = X@beta.value
```

```
        exercisedPayoffs.insert(0, payOffs)
        exercisedExpDisPayoffs.insert(0, discountedExpectedPayoffs)
    exercisedPayoffs = np.array(exercisedPayoffs).T
    exercisedExpDisPayoffs = np.array(exercisedExpDisPayoffs).T

    discountedPayoffsMax = []
    for i in range(nSimulations):
        for j in range(nSteps):
            exerPayoff = exercisedPayoffs[i][j]
            exerDisPayoff = exercisedExpDisPayoffs[i][j]
            if exerPayoff > exerDisPayoff :
                disPayoff = exerPayoff*np.exp(-r*step*j)
                break
            else:
                disPayoff = exerDisPayoff*np.exp(-r*step*j)
        discountedPayoffsMax.append(disPayoff)

    avg = np.mean(discountedPayoffsMax)
    var = np.var(discountedPayoffsMax, ddof=1)

    return(discountedPayoffsMax, avg, var)
```

6 Control Variates Implementation

The method of control variates is a variance reduction method associated with Monte Carlo simulation. We are interested in applying the control variate method to produce a better estimate of the American put option price for GOOG. The method of control variates works by taking in a random variable Y whose mean, μ_y, we wish to estimate, and another random variable X whose mean, μ_x, we estimated. We want to use our knowledge of $\mathbb{E}(X)$ to improve the estimation of $\mathbb{E}(Y)$. To do so, we first let (X_i, Y_i), $i = 1, 2, ..., n$ be independently and identically distributed draws of (X, Y), but X and Y are not necessarily independent. With $\hat{Y} = \bar{Y} + \beta(\mu_x - \bar{X})$, $\bar{X} = \frac{X_1+\cdots+X_n}{n}$, and $\bar{Y} = \frac{Y_1+\cdots+Y_n}{n}$, we define

$$\text{Var}(\hat{Y}) = \frac{\sigma_Y^2}{n} + \beta^2\frac{\sigma_X^2}{n} - 2\beta\text{corr}(X,Y)\frac{\sigma_X\sigma_Y}{n}.$$

Setting $\beta = \frac{\sigma_Y}{\sigma_X}\text{corr}(X,Y)$ gives us

$$\text{Var}(\bar{Y}) = \frac{\sigma_Y^2}{n}[1 - \text{corr}^2(X,Y)].$$

Since we do not know $\text{corr}(X,Y), \sigma_X$, and σ_Y, we must estimate them using:

$$\sigma_X^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2,\ \sigma_{XY}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y}).$$

Then, let $\beta = \frac{\sigma_{XY}^2}{\sigma_X^2}$. We implemented this using the following Python code below:

```
def contvariate(x,mu,y):

    c = -np.cov(x,y)[1,0]/np.var(x,ddof=1)

    return(np.mean(y+c*(x-mu)))
```

7 Conclusion

After compiling our Python codes, we gathered that with the historical price data from one year from June 12th, 2020, 100 simulations, a one year libor yield of 0.63%, and an initial stock price of $1413.18, the generated American Put option price without control variates for GOOG with strike price $K = 1200$ is $114.74 with a variance of 2.25. Comparing this with the European Put option price, we got $104.42 as the premium with a much higher variance of 627.58. After applying the control variate method, our generated American Put option premium was $114.75 which is slightly higher than the original generated premium without the control variate. This tells us that the implementation of both the Monte Carlo and control variate are a better approximation in determining American Put Option Prices since it not only takes into consideration of potential future asset paths but also minimizes the number of simulation paths required to generate an accurate option pricing, as opposed to being priced similarly to a European Put option. Thus, we concluded that $114.75 would be a fair put option price for GOOG with strike price $1200 and expiration date of June 18th, 2021.

Appendix: Additional Python Code

```
# import packages
# !pip3 install pandas_datareader
# !pip3 install yfinance
# !pip3 install cvxpy
import numpy as np
import matplotlib as m
import pandas as pd
import pandas_datareader as pdr
from pandas_datareader import data
import yfinance as yf
import datetime as dt
from datetime import date
import scipy.stats as stats
import cvxpy as cp
yf.pdr_override()

#USE    : To get historical data for a particular stock
#INPUT  : ticker of the company
#OUTPUT : array of close price of data
def GetData(companyName, years=1):

    today = date.today()
    # Setting start date to historical years
    startDate = today.replace(today.year-years)
    # Date : Open High Low Close "Adj Close" Volume
    data = pdr.get_data_yahoo(companyName, startDate, today)
    # Cleaning to get the close prices only
    closePrice = [data["Close"][i] for i in range(len(data))]

    return closePrice

#USE    : To get continuous annual dividend yield for a stock
#INPUT  : ticker of the company, number of dividend payments in a year
#OUTPUT : continuous annual dividend yield
def ContDiv(companyName):

    stock = yf.Ticker(companyName)
    # Get current stock price
    S0 = pdr.get_data_yahoo(companyName).iloc[-1,:]['Close']

    if len(stock.dividends) > 0:
        # get frequency of dividend payments in a year
        div_df = stock.actions[['Dividends']]
        div_df = div_df[div_df['Dividends'] > 0]
        freq = len(div_df['2019'])
```

```
        # compute annual dividend yield
        annualDY = (stock.dividends[-1]*freq)/S0
    else:
        annualDY = 0
    # converts annual dividend yield to effective continuous yield
    delta = np.log(annualDY + 1)

    return delta

"""
compute true Black-Scholes price of the option
define function for computing Black-Scholes price for Call or Put
inputs: S0 = initial stock price, K = strike price, r = risk-free interest rate,
        T = maturity time, sig = sigma (volatility),
        type = 'C' for Call or 'P' for Put
"""
def BlackScholes(S0, K, r, T, sigma, optionType):
    if optionType=="C":
        d1 = (np.log(S0/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
        d2 = d1 - sigma*np.sqrt(T)

        value = S0*stats.norm.cdf(d1,0,1) - K*np.exp(-r*T)*stats.norm.cdf(d2,0,1)

    elif optionType=="P":
        d1 = (np.log(S0/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
        d2 = d1 - sigma*np.sqrt(T)

        value =  (K*np.exp(-r*T)*stats.norm.cdf(-d2, 0, 1) -
                                S0*stats.norm.cdf(-d1,0,1))

    return(value)
```