ACM 650

Final Project

Multi-period binomial tree pricing model

12/04/18

Harsha Kankanamge

Abstract

Build a multi-period binomial tree option pricing model to estimate the value of a call option price and put option price using Cox-Ross-Rubinstein model and Lognormal model. And do the comparison for output of those two model.

Introduction

Once we have understood the one period binomial model it is very easy to extend the two period model or slightly more period binomial model. But extend to thousand or more period binomial model is not easy to do by hand. Binomial model can produce a very good approximation when it is extended to a sufficiently great number of periods. In this project we consider how to extend the binomial model to thousands of periods. So we used Python program to do multi-period binomial model.

For this project, we decided to use the stock prices for a Japanese based company, “Nintendo” (NTDOY). We chose to work with the coding language of Python, due to its ability to pull data sets and solve what is needed. We pulled directly from Yahoo Finance with the following code:

```
ntdoy = pdr.get_data_yahoo('NTDOY',
                           start=datetime.datetime(1996, 11, 18),
                           end=datetime.datetime(2018, 11, 27))
```

This code pulls all stock data from Yahoo Finance from the beginning of the stock (November 18, 1996) up to twenty-two years later (till November 27, 2018).

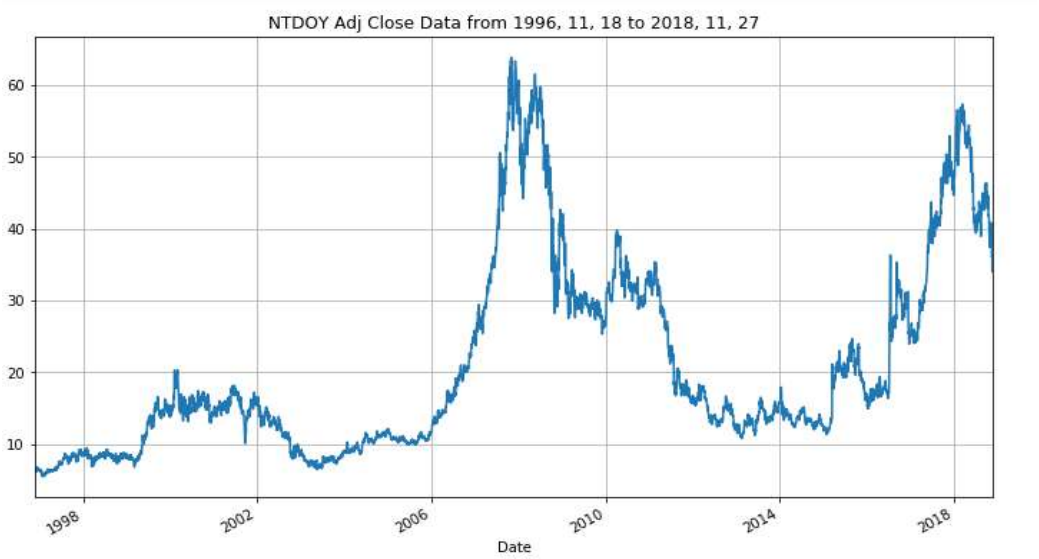
Next, using data provided, we assigned the current price to be equal to the adjusted close price on the last date (giving us $37.5). We also solved for the little delta by taking the average of all the adjusted close prices with the dividend price at that time (giving us a result of 0.0112444).

```
Current_price=37.5
Ldelta=.011244408
```

We wanted to see a visualization of all the data prices given. Using the following code, we were able to graph a plot of all the ‘Adj close’ prices:

```
ntdoy['Adj Close'].plot(grid=True)
plt.title('NTDOY Data from 1996, 11, 18 to 2018, 11, 27')
plt.show()
```

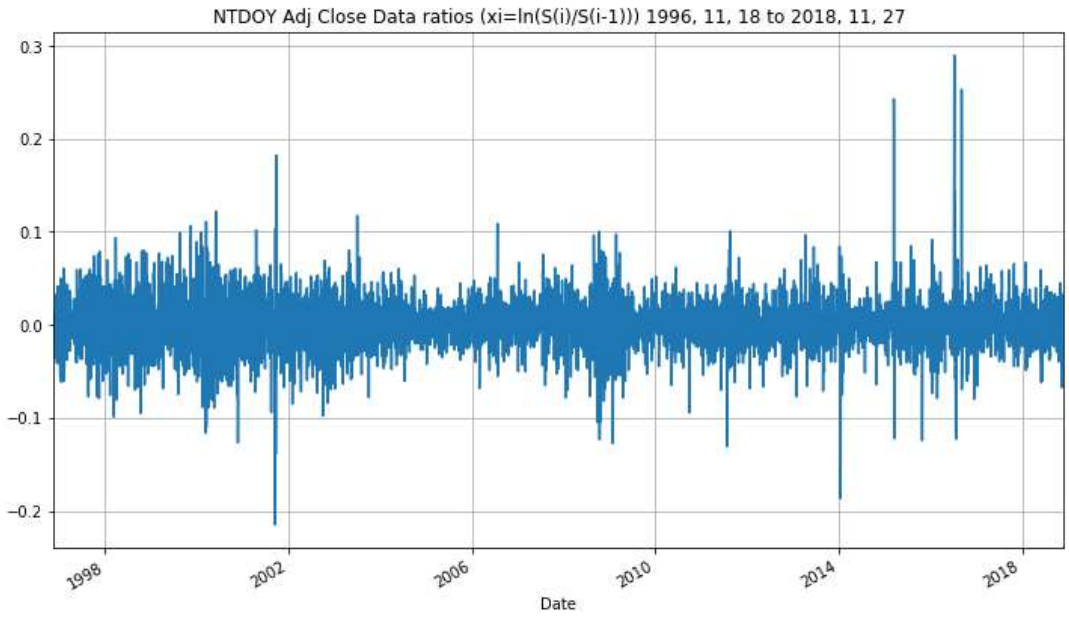
Resulting in:



Using the data directly from Yahoo Finance, we were able to pull out the necessary data to calculate all of need X_i's.

```
class stock_vol:


    def __init__(self, tk, start, end):
        self.tk = tk
        self.start = start
        self.end = end
        all_data = pdr.get_data_yahoo(self.tk, start=self.start, end=self.end)
        self.stock_data = pd.DataFrame(all_data['Adj Close'], columns=["Adj Close"])
        self.stock_data["log"] = np.log(self.stock_data)-np.log(self.stock_data.shift(1))


    def mean_sigma(self):
        st = self.stock_data["log"].dropna().ewm(span=252).std()
        sigma = st.iloc[-1]
        return sigma


    def garch_sigma(self):
```

```
            model = arch.arch_model(self.stock_data["log"].dropna(), mean='Zero',
vol='GARCH', p=1, q=1)
            model_fit = model.fit()
            forecast = model_fit.forecast(horizon=1)
            var = forecast.variance.iloc[-1]
            sigma = float(np.sqrt(var))
            return sigma


if __name__ == "__main__":
    vol = stock_vol("NTDOY", start="1996-11-18", end="2018-11-27")
    test = vol.stock_data["log"].dropna()
    print(test)
    fig = plot_acf(test)
    plt.show()


test.plot(grid=True)
plt.title('NTDOY Adj Close Data ratios (xi=ln(S(i)/S(i-1))) 1996, 11, 18 to 2018, 11,
27')
plt.show()
```



Using the data above, we were now able to solve for the Volatility.

```
import math as m
xi=[]
for i in range(0,5543):
    xi.append(test[i])
x1=0
for i in xi:
    x1=x1+i**2
x2=x1/len(xi)
mean=sum(xi)/len(xi)
mean=mean**2
volatility=m.sqrt(252)*m.sqrt(len(xi)*(x2-mean)/(len(xi)-1))
print('Volatility =', volatility)
```

Which gave us: Volatility = 0.43905915864868805

We are able to now build an example stock tree from the data. First thing we need to find is what the values of ‘u’ and ‘d’ are. We did that with the following Python code:

```
import sys
import numpy as np
from math import pow
import math
# create the Binomial Tree

def build_stock_tree(S,T,N,sig):
    dt=T/N;
    u=math.exp(sig*math.sqrt(dt));
    d=math.exp(-sig*math.sqrt(dt));
    tree=np.zeros((N+1,N+1))
```

```
    for i in range(N+1):
        for j in range(i+1):
            tree[i][j]=S*pow(u,j)*pow(d,i-j)

    return(tree)
```

With the function giving us ‘u’ and ‘d’ we can create our example tree model as a good visualization of what we are solving:

```
s = [[str(e) for e in row] for row in build_stock_tree(Current_price,0.5,4,volatility)]
lens = [max(map(len, col)) for col in zip(*s)]
fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)
table = [fmt.format(*row) for row in s]
print('\n'.join(table))
```

37.0				
31.680047250334702	43.21331938624355			
27.125010642795658	37.00000000000001	50.470026280472766		
23.224908617016165	31.680047250334706	43.21331938624355	58.945334191626344	
19.885573037075073	27.12501064279566	37.00000000000001	50.470026280472766	68.8438798040976

First we decided we wanted to see what the Cox-Ross-Rubinstein Tree values come out to be. We used the following code to solve for the math involved:

```
import numpy as np
def CRRTree(type,S0, K, r, sigma,Ldelta, T, N):

#calculate delta T
```

```
        deltaT = float(T) / N

# up and down factor will be constant for the tree so we calculate outside the loop
        u = np.exp(sigma * np.sqrt(deltaT))
        d = np.exp(-sigma * np.sqrt(deltaT))

        #to work with vector we need to init the arrays using numpy
        fs =    np.asarray([0.0 for i in range(N + 1)])

#we need the stock tree for calculations of expiration values
        fs2 = np.asarray([(S0 * u**j * d**(N - j)) for j in range(N + 1)])

#we vectorize the strikes as well so the expiration check will be faster
        fs3 =np.asarray( [float(K) for i in range(N + 1)])

        a = np.exp(r * deltaT)
        p=((np.exp((r-Ldelta)*deltaT))-d)/(u-d);
        oneMinusP = 1.0 - p

# Compute the leaves, f_{N, j}
        if type =="C":
            fs[:] = np.maximum(fs2-fs3, 0.0)
        else:
            fs[:] = np.maximum(-fs2+fs3, 0.0)

#calculate backward the option prices
        for i in range(N-1, -1, -1):
            fs[:-1]=np.exp(-r * deltaT) * (p * fs[1:] + oneMinusP * fs[:-1])
            fs2[:]=fs2[:]*u
```

```
# print fs
    return fs[0]
```

Using all of the values we solved we can now determine the put and call options for the Cox-Ross-Rubinstein model. The following code displays the first 9 values given in each option.

Put option:

```
for i in range(1,10):
    print(CRRTree('p',Current_price, Current_price, 0.0221,volatility,Ldelta, 0.5, N=i))
```

Which prints out the following:

5.550985732987845

3.912526904981438

4.8074254771544185

4.155890706828922

4.654228769685962

4.243637755686113

4.588847540998152

4.288644450868489

4.552686527038348

As well as the 100,000 through 100,010th values:

```
for i in range(100000,100010):
    print(CRRTree('P',Current_price, Current_price, r,volatility,Ldelta, 0.5, N=i))
```

Which give us:

4.427562475493936
4.4275849231754885
4.427562475728777

4.427584922662847
4.427562476047598
4.427584922537167
4.427562476022619
4.427584922495112
4.427562476532915
4.427584922184095

Call option:

```
for i in range(1,10):
    print(CRRTree('C',Current_price, Current_price, 0.0221,volatility,Ldelta, 0.5, N=i))
```

Which prints out the following:

5.75015329548223

4.111694467475819

5.006593039648809

4.355058269323318

4.853396332180331

4.442805318180507

4.788015103492518

4.487812013362878

4.751854089532722

And the 100,000 - 100,010 values:

```
for i in range(100000,100010):
    print(CRRTree('C',Current_price, Current_price, r,volatility,Ldelta, 0.5, N=i))
```

Giving us the following:

4.6267300380144585

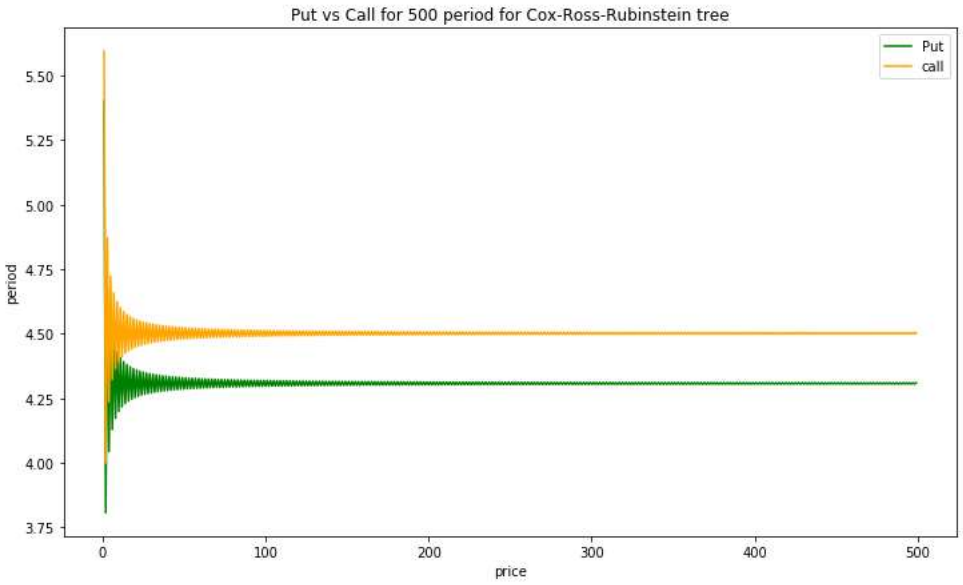
4.6267524852964295
4.626730038228505
4.6267524855004325
4.626730038366811
4.626752485113659
4.626730038816596
4.626752484651023
4.626730038829164
4.626752484577381

For a visual representation to the data we solved for, we needed to plot every point. It does converge within the first 200 or less steps, however we decided to plot up to the first 500 steps.

```
%matplotlib inline

import matplotlib.pyplot as plt

CRR_p=[]

CRR_c=[]

for i in range(5,500):

    CRR_p.append(CRRTree('p',Current_price, Current_price,
0.0221,volatility,Ldelta, 0.5, N=i))

for i in range(5,500):

    CRR_c.append(CRRTree('C',Current_price, Current_price,
0.0221,volatility,Ldelta, 0.5, N=i))



plt.rcParams['figure.figsize'] = (12.0, 7.0)


plt.plot(range(5,500), CRR_p, c='g', label='Put')

plt.plot(range(5,500), CRR_c, c='orange', label='call')

plt.xlabel('period')

plt.ylabel('period')

plt.title('Put vs Call for 500 period for Cox-Ross-Rubinstein tree')

plt.legend()
```

```
plt.show()
```

Resulting in the following graph:



As you can tell from the resulting graph, the data starts to converge right around 100. It continues to stay the same for the rest of the results. This is expected with any data of this type.

Now, we wanted to see what the Log-Normal tree looks like when computed with the same data.

The following code was used to solve the math needed to complete the log-normal model:

```
import numpy as np

def LogNormalTree(type,S0, K, r, sigma,Ldelta, T, N):


#calculate delta T

    deltaT = float(T) / N
```

```
# up and down factor will be constant for the tree so we calculate outside the loop

    u = np.exp((r-Ldelta-(1/2)*sigma*sigma)*deltaT+sigma*np.sqrt(deltaT));
    d = np.exp((r-Ldelta-(1/2)*sigma*sigma)*deltaT-sigma*np.sqrt(deltaT));

#to work with vector we need to init the arrays using numpy
    fs =    np.asarray([0.0 for i in range(N + 1)])

#we need the stock tree for calculations of expiration values
    fs2 = np.asarray([(S0 * u**j * d**(N - j)) for j in range(N + 1)])

#we vectorize the strikes as well so the expiration check will be faster
    fs3 =np.asarray( [float(K) for i in range(N + 1)])

    a = np.exp(r * deltaT)
    p=((np.exp((r-Ldelta)*deltaT))-d)/(u-d);
    oneMinusP = 1.0 - p

# Compute the leaves, f_{N, j}
    if type =="C":
        fs[:] = np.maximum(fs2-fs3, 0.0)
    else:
        fs[:] = np.maximum(-fs2+fs3, 0.0)

#calculate backward the option prices
    for i in range(N-1, -1, -1):
        fs[:-1]=np.exp(-r * deltaT) * (p * fs[1:] + oneMinusP * fs[:-1])
        fs2[:]=fs2[:]*u
```

```
# print fs
    return fs[0]
```

Using all of the values we solved we can now do the same thing we did for the Cox-Ross-Rubinstein model and determine the put and call options for the Log-Normal model. The following code displays the first 9 values given in each option.

Put option:

```
for i in range(1,10):
print(LogNormalTree('p',Current_price, Current_price, 0.0221,volatility,Ldelta, 0.5, N=i))
```

Which prints out the following:

5.4311071819904395

4.25760415424375

4.742222911588577

4.403133730123513

4.598560494055691

4.442254621233452

4.537110552681642

4.456954521878634

4.503092720593405

And again, the 100,000 through 100,009 values:

```
for i in range(100000,100010):
    print(LogNormalTree('p',Current_price, Current_price, r,volatility,Ldelta, 0.5, N=i))
```

Resulting in:

4.4275773968652965
4.427580074757706
4.427577385880092
4.427580082922271
4.427577374946007
4.427580091538743
4.42757736366065
4.427580100155761
4.427577352876826
4.427580108519727

Call option:

```
for i in range(1,10):
print(LogNormalTree('C',Current_price, Current_price, 0.0221,volatility,Ldelta, 0.5, N=i))
```

Which prints out the following:

5.630274744484825

4.456771716738132

4.9413904740829695

4.602301292617906

4.797728056550064

4.641422183727843

4.736278115176005

4.656122084373023

4.702260283087781

Along with the 100,000 - 100,009th values:

```
for i in range(100000,100010):
    print(LogNormalTree('C',Current_price, Current_price, r,volatility,Ldelta, 0.5, N=i))
```

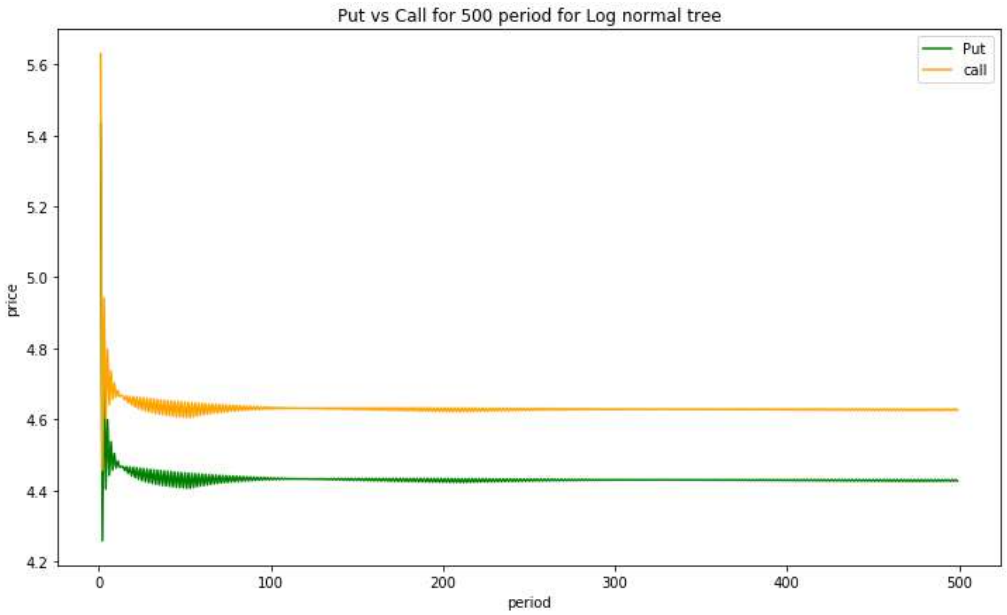
Which prints out:

4.6267449593851735
4.62674763687875
4.626744948379529
4.626747645759001
4.626744937265348
4.626747654116349
4.62674492645451
4.62674766231179
4.626744915172341
4.626747670913005

For a visual representation to the data we solved for, we needed to plot every point. It does converge within the first 200 or less steps, however we decided again to plot up to the first 500 steps.

```
Ln_p=[]

Ln_c=[]

for i in range(5,500):

    Ln_p.append(LogNormalTree('p',Current_price, Current_price,
0.0221,volatility,Ldelta, 0.5, N=i))

for i in range(5,500):

    Ln_c.append(LogNormalTree('C',Current_price, Current_price,
0.0221,volatility,Ldelta, 0.5, N=i))

plt.plot(range(5,500), Ln_p, c='g', label='Put')

plt.plot(range(5,500), Ln_c, c='orange', label='call')

plt.xlabel('period')

plt.ylabel('period')

plt.title('Put vs Call for 500 period for Log normal tree')

plt.legend()

plt.show()
```

Resulting in the following graph:

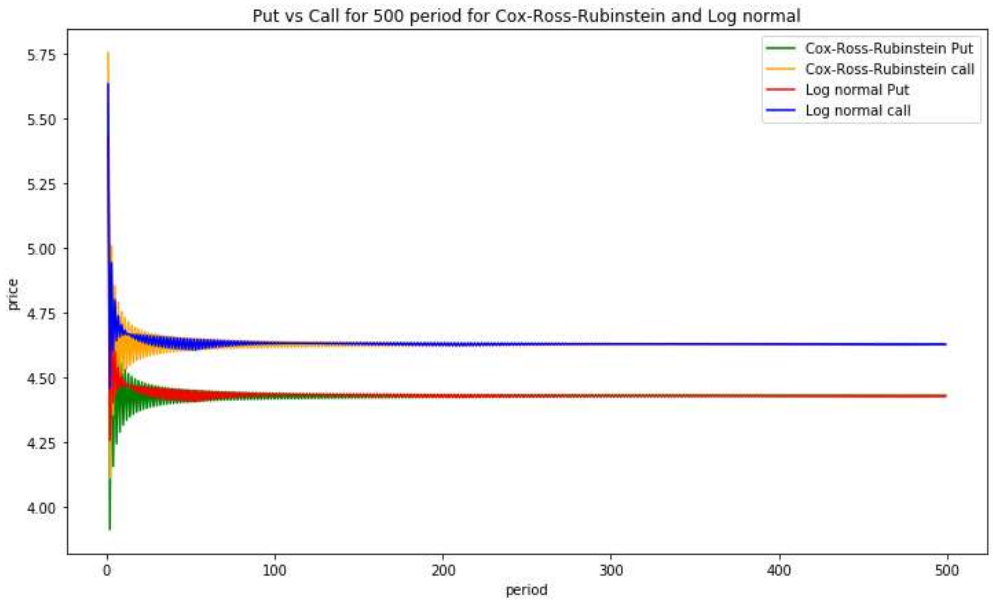


Based on the above graph, it looks like the points do not converge till a little later than in the Cox-Ross-Rubinstein model but still right around 100. It gets very small around 150 (where it converges in the Cox-Ross-Rubinstein model) but then gets a bit larger around 200. After that, it seems to be the same for all remaining points. This could be a result in just errors being factored in. As the number of periods increase, the greater the chance of error occurring comes into play. Either way, this still looks like an accurate graph to the model that we are looking for.

We then wanted to compare the two models with each other. We did this by placing the two graphs on top of each other.

```
plt.plot(range(1,500), CRR_p, c='g', label='Cox-Ross-Rubinstein Put')
plt.plot(range(1,500), CRR_c, c='orange', label='Cox-Ross-Rubinstein call')
plt.plot(range(1,500), Ln_p, c='red', label='Log normal Put')
plt.plot(range(1,500), Ln_c, c='blue', label='Log normal call')
plt.xlabel('period')
plt.ylabel('price')
plt.title('Put vs Call for 500 period for Cox-Ross-Rubinstein and Log normal')
plt.legend()
plt.show()
```

Giving us the graph:



Conclusions

As you can see, the both of the models are converging into the same values at about the same time. This shows that each of the models give the same data and are accurate to each other.

After about 150 sub-periods, option prices converge to final values.

Before the 150 sub-period, CRR model has high variance for option price than Lognormal model.

Finally, we want to know the call option price for lognormal model at the point where the graph converges to one value. For that, we looked at the option at point 150 resulting in:

```
print(LogNormalTree('C',Current_price, Current_price, 0.0221,volatility,Ldelta, 0.5, N=150))
```

Answer: 4.626750850275347