March 2, 2023

0.1 DERIVATIVE PRICING

1 MERTON (1976) MODEL

Kristof Kassa

---

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as ss
```

1.1 1. Implementing the Merton Model

We will start by implementing the Merton model in Python. The model has the following SDE:

$$dS_t = (r - r_j)\, S_t dt + \sigma S_t dZ_t + J_t S_t dN_t$$

with the following discretized form:

$$S_t = S_{t-1}\left(e^{\left(r - r_j - \frac{\sigma^2}{2}\right)dt + \sigma\sqrt{dt}z_t^1} + \left(e^{\mu_j + \delta z_t^2} - 1\right) y_t\right)$$

where z_t^1 and z_t^2 follow a standard normal and y_t follows a Poisson process. Finally, r_j equals to:

$$r_j = \lambda\left(e^{\mu_j + \frac{\delta^2}{2}}\right) - 1$$

In order to obtain the parameters of the model, we will perform an exercise of calibration to option market prices. Let's assume these parameters as given and equal to:

[2]:
```
lamb = 0.75  # Lambda of the model
mu = -0.6  # Mu
delta = 0.25  # Delta
```

Let's also assume the following information for the current stock price, number of simulations in our Monte Carlo estimations, etc.

[3]:
```
r = 0.05  # Risk-free rate
sigma = 0.2  # Volatility
T = 1.0  # Maturity/time period (in years)
S0 = 100  # Current Stock Price

Ite = 10000  # Number of simulations (paths)
M = 50  # Number of steps
dt = T / M  # Time-step
```

Next, we will calculate the random numbers that we need, together with some variable definitions for later on:

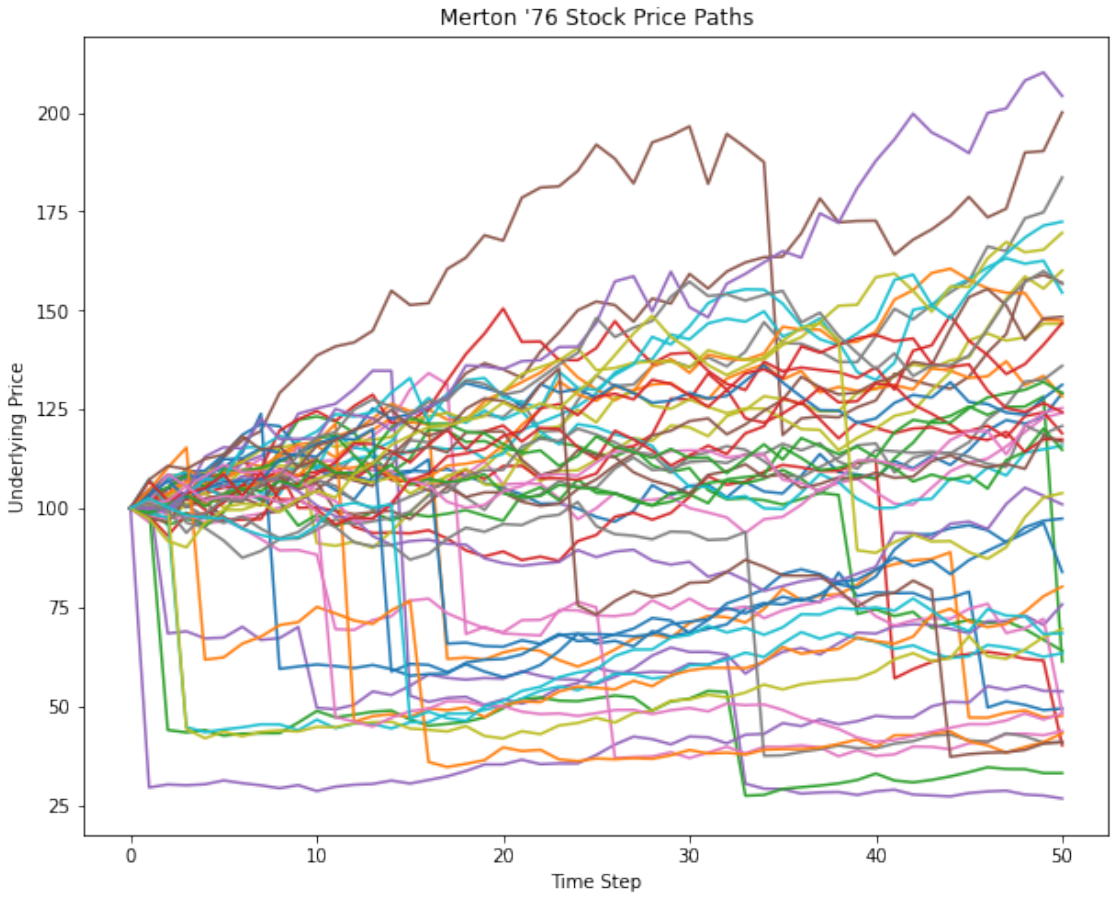
[4]:
```
SM = np.zeros((M + 1, Ite))
SM[0] = S0

# rj
rj = lamb * (np.exp(mu + 0.5 * delta**2) - 1)

# Random numbers
z1 = np.random.standard_normal((M + 1, Ite))
z2 = np.random.standard_normal((M + 1, Ite))
y = np.random.poisson(lamb * dt, (M + 1, Ite))
```

With this info, and using the Monte Carlo methods we are familiar with already, we can simulate paths for our stock price under Merton SDE:

[5]:
```
for t in range(1, M + 1):
    SM[t] = SM[t - 1] * (
        np.exp((r - rj - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z1[t])
        + (np.exp(mu + delta * z2[t]) - 1) * y[t]
    )
    SM[t] = np.maximum(
        SM[t], 0.00001
    )  # To ensure that the price never goes below zero!
```

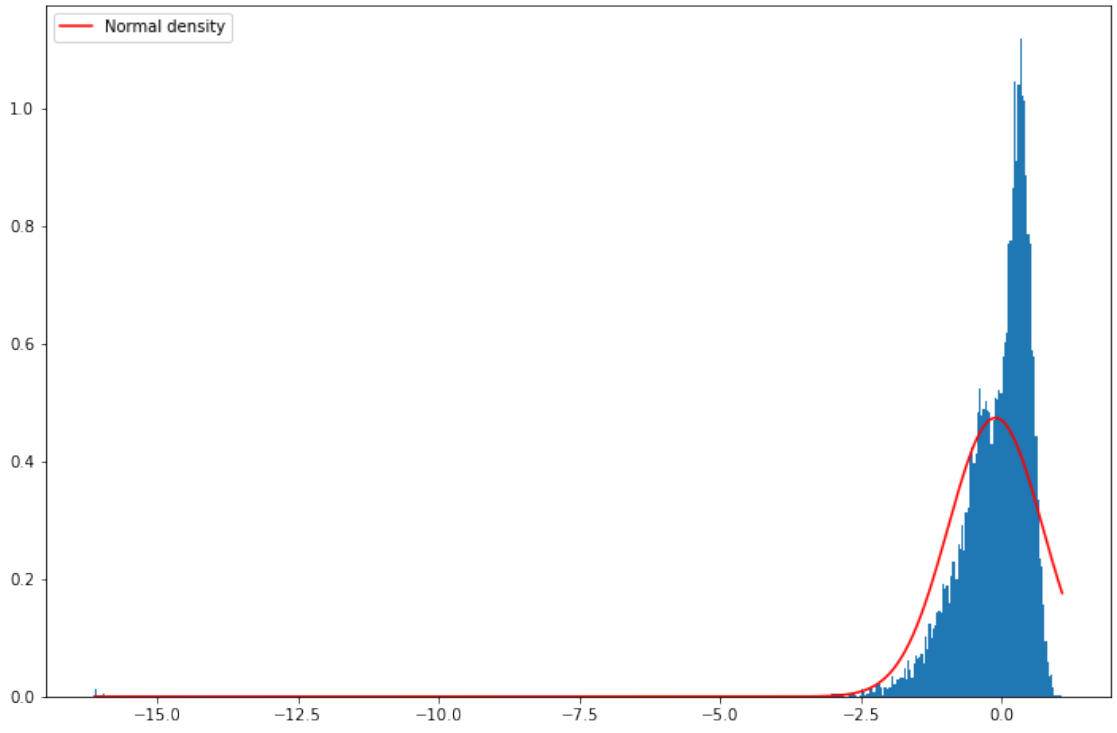
Let's see how stock price evaluation looks for a sample of 50 paths:

[6]:
```
plt.figure(figsize=(10, 8))
plt.plot(SM[:, 100:150])
plt.title("Merton '76 Stock Price Paths")
plt.xlabel("Time Step")
plt.ylabel("Underlying Price")
plt.show()
```



Distribution of stock returns that this model produces:

[7]:
```
retSM = np.log(SM[-1, :] / S0)
x = np.linspace(retSM.min(), retSM.max(), 500)

plt.figure(figsize=(12, 8))
plt.hist(retSM, density=True, bins=500)
plt.plot(
    x, ss.norm.pdf(x, retSM.mean(), retSM.std()), color="r", label="Normal␣
 ↪density"
)
plt.legend()
plt.show()
```



The distribution generated by the Merton model has more kurtosis than a normal distribution and a negative skewness, consistent with the different stylized facts of stock returns. We will, later on, discuss how changing the different parameters modifies this distribution. First, let's compare this result to the returns produced by GBM and Heston.

1.2 2. Comparison to Heston and GBM Returns

Next, we will compare the return (and its distribution) generated by other models like Black-Scholes (GBM) or Heston.

Let's start with the former and, when possible, use the same parameters as in the previous example.

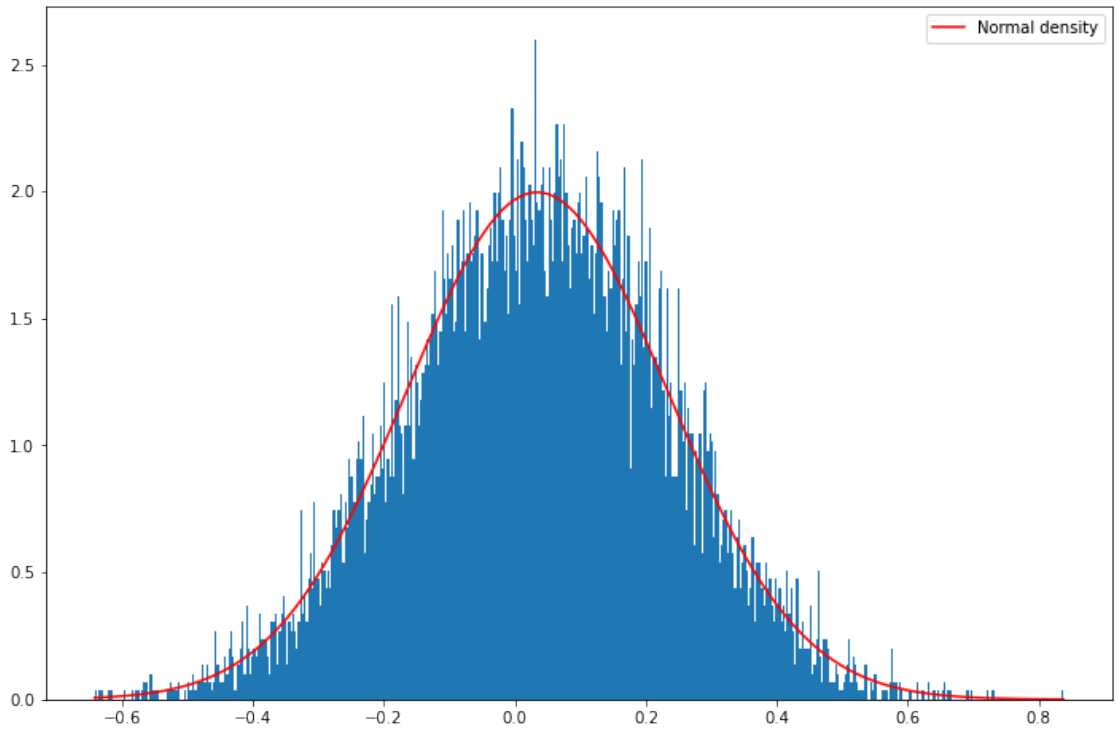
1.2.1 2.1. GBM Returns and Distribution

Let's implement the classic GBM and check the distribution of its returns:

[8]:
```
S = np.zeros((M + 1, Ite))
S[0] = S0
for t in range(1, M + 1):
    S[t] = S[t - 1] * np.exp(
        (r - 0.5 * sigma**2) * dt
        + sigma * np.sqrt(dt) * np.random.standard_normal(Ite)
    )
```

[9]:
```
retS = np.log(S[-1, :] / S0)
y = np.linspace(retS.min(), retS.max(), 500)
```

[10]:
```
plt.figure(figsize=(12, 8))
plt.hist(retS, density=True, bins=500)
plt.plot(y, ss.norm.pdf(y, retS.mean(), retS.std()), color="r", label="Normal␣
↪density")
plt.legend()
plt.show()
```



Obviously, what we get here is the classic normal distribution.

1.3 2.2. Heston Returns and Their Distribution

Finally, let's compare this to Heston-produced returns:

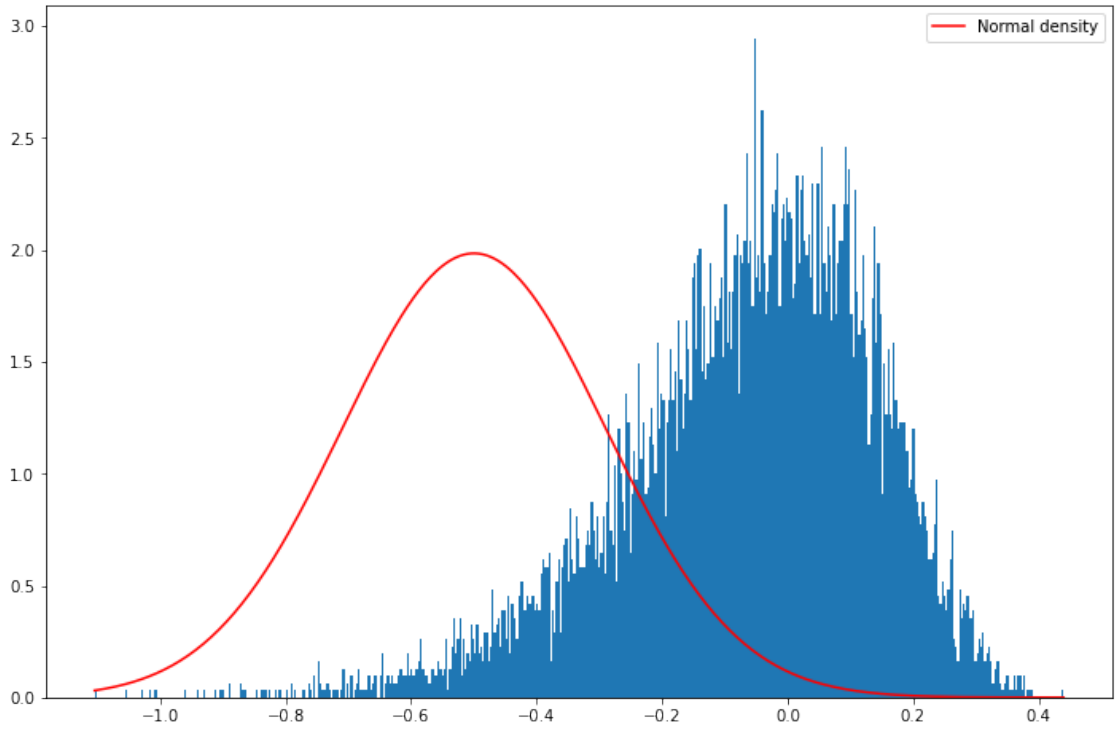
[11]:
```
def SDE_vol(v0, kappa, theta, sigma, T, M, Ite, rand, row, cho_matrix):
    dt = T / M  # T = maturity, M = number of time steps
    v = np.zeros((M + 1, Ite), dtype=np.float)
    v[0] = v0
    sdt = np.sqrt(dt)  # Sqrt of dt
    for t in range(1, M + 1):
        ran = np.dot(cho_matrix, rand[:, t])
        v[t] = np.maximum(
```

```
            0,
            v[t - 1]
            + kappa * (theta - v[t - 1]) * dt
            + np.sqrt(v[t - 1]) * sigma * ran[row] * sdt,
        )
    return v


def Heston_paths(S0, r, v, row, cho_matrix):
    S = np.zeros((M + 1, Ite), dtype=float)
    S[0] = S0
    sdt = np.sqrt(dt)
    for t in range(1, M + 1, 1):
        ran = np.dot(cho_matrix, rand[:, t])
        S[t] = S[t - 1] * np.exp((r - 0.5 * v[t]) * dt + np.sqrt(v[t]) *␣
↪ran[row] * sdt)

    return S


def random_number_gen(M, Ite):
    rand = np.random.standard_normal((2, M + 1, Ite))
    return rand
```

[12]:
```
# Heston given parameters
v0 = 0.04
kappa_v = 2
sigma_v = 0.2
theta_v = 0.04
rho = -0.9


# Generating random numbers from standard normal
rand = random_number_gen(M, Ite)


# Covariance Matrix
covariance_matrix = np.zeros((2, 2))
covariance_matrix[0] = [1.0, rho]
covariance_matrix[1] = [rho, 1.0]
cho_matrix = np.linalg.cholesky(covariance_matrix)

# Volatility process paths
V = SDE_vol(v0, kappa_v, theta_v, sigma_v, T, M, Ite, rand, 1, cho_matrix)

# Underlying price process paths
HS = Heston_paths(S0, r, V, 0, cho_matrix)
```

```
/tmp/ipykernel_823/186567487.py:3: DeprecationWarning: `np.float` is a
deprecated alias for the builtin `float`. To silence this warning, use `float`
by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  v = np.zeros((M + 1, Ite), dtype=np.float)
```

[13]:
```
retHS = np.log(HS[-1, :] / S0)
q = np.linspace(retHS.min(), retHS.max(), 500)


plt.figure(figsize=(12, 8))
plt.hist(retHS, density=True, bins=500)
plt.plot(
    q, ss.norm.pdf(y, retHS.mean(), retHS.std()), color="r", label="Normal␣
 ↪density"
)
plt.legend()
plt.show()
```

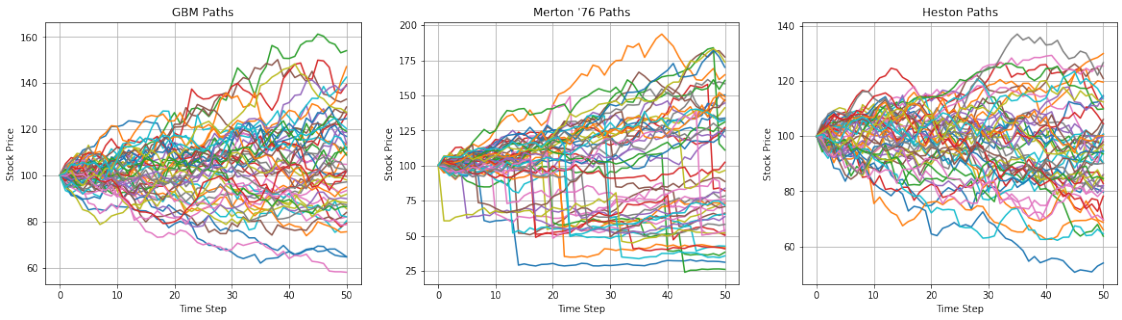


This is the distribution of returns produced by Heston.

These clear differences between the different models are also evident just by looking at the kind of stock price paths they produce:

[14]:
```
fig = plt.figure(figsize=(20, 5))
ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)

ax1.plot(S[:, :50])
ax1.grid()
ax1.set_title("GBM Paths")
ax1.set_ylabel("Stock Price")
ax1.set_xlabel("Time Step")

ax2.plot(SM[:, :50])
ax2.grid()
ax2.set_title("Merton '76 Paths")
ax2.set_ylabel("Stock Price")
ax2.set_xlabel("Time Step")

ax3.plot(HS[:, :50])
ax3.grid()
ax3.set_title("Heston Paths")
ax3.set_ylabel("Stock Price")
ax3.set_xlabel("Time Step")
```

[14]:
```
Text(0.5, 0, 'Time Step')
```



1.4 3. Option Pricing under the Different Models

Next, let's explore the consequences of these differences in option pricing. For simplicity, let's assume a European call option with $K = 90$ and $T = 1$ year. Let's use Monte Carlo methods for this and assume the parameters from before.

1.4.1 3.1. Pricing under Black-Scholes (GBM)

[15]:
```
def bs_call_mc(S, K, r, sigma, T, t, Ite):

    data = np.zeros((Ite, 2))
    z = np.random.normal(0, 1, [1, Ite])
    ST = S * np.exp((T - t) * (r - 0.5 * sigma**2) + sigma * np.sqrt(T - t) * z)
    data[:, 1] = ST - K

    average = np.sum(np.amax(data, axis=1)) / float(Ite)

    return np.exp(-r * (T - t)) * average
```

[16]:
```
print("European Call Price under BS (MC): ", bs_call_mc(S0, 90, r, sigma, T, 0,␣
↪Ite))
```

```
European Call Price under BS (MC):  16.687401848182724
```

3.2 Pricing under Heston

[17]:
```
def heston_call_mc(S, K, r, T, t):
    payoff = np.maximum(0, S[-1, :] - K)

    average = np.mean(payoff)

    return np.exp(-r * (T - t)) * average
```

[18]:
```
print("European Call Price under Heston: ", heston_call_mc(HS, 90, r, T, 0))
```

```
European Call Price under Heston:  10.067275108387355
```

3.3 Pricing under Merton

[19]:
```
def merton_call_mc(S, K, r, T, t):
    payoff = np.maximum(0, S[-1, :] - K)

    average = np.mean(payoff)

    return np.exp(-r * (T - t)) * average
```

[20]:
```
print("European Call Price under Merton: ", merton_call_mc(SM, 90, r, T, 0))
```

```
European Call Price under Merton:  27.798840733988175
```

Are these results for the price of the options consistent with the distributions we observed?

1.5 4. Discussion of Merton Model Parameters

Finally, let's discuss empirically how the different parameters from Merton influence the type and form of the distribution of stock returns. Specifically, we will focus on the parameters λ and μ_j:

- $\lambda \rightarrow$ Intensity (frequency) of the jump (shock) to stock prices

- $\mu_j \rightarrow$ Average jump size (can be positive or negative)

As we have already mentioned more than a few times, the actual values that we will use for these parameters will come from a calibration to observed option market prices.

While we will tackle this issue soon, for now, it will be helpful to understand what the sign and magnitude of these parameters mean from an empirical standpoint.

1.5.1 4.1. Changing λ

In the previous example, we used a $\lambda = 0.75$, which accounted for the intensity (frequency) of the jumps in interval t. In this case, we expect 0.75 events in a year on average. In other words, the average rate at which jumps occur is 0.75 (75%).

What will happen to the previous distribution of returns if we increase (or decrease) this number?

Let's check it by imposing $\lambda = 0.99$:

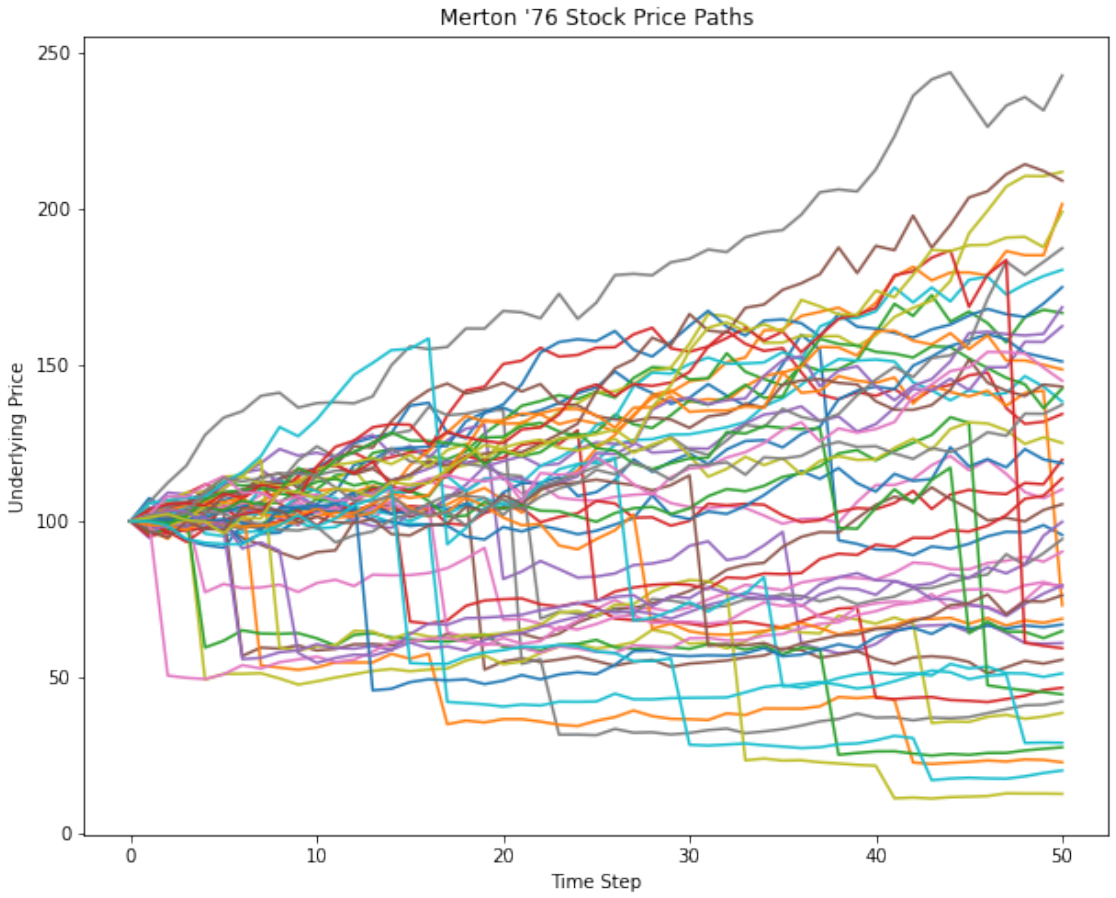
[21]:
```
lamb = 0.75  # Lambda of the model
mu = 0.6  # Mu
delta = 0.25  # Delta

r = 0.05  # Risk-free rate
sigma = 0.2  # Volatility
T = 1.0  # Maturity/time period (in years)
S0 = 100  # Current Stock Price

Ite = 10000  # Number of simulations (paths)
M = 50  # Number of steps
dt = T / M  # Time-step

SM = np.zeros((M + 1, Ite))
SM[0] = S0

# rj
rj = lamb * (np.exp(mu + 0.5 * delta**2) - 1)

# Random numbers
z1 = np.random.standard_normal((M + 1, Ite))
z2 = np.random.standard_normal((M + 1, Ite))
y = np.random.poisson(lamb * dt, (M + 1, Ite))
```

[22]:
```
lamb = 0.99  # Lambda of the model
mu = -0.6  # Mu
delta = 0.25  # Delta

r = 0.05  # Risk-free rate
sigma = 0.2  # Volatility
T = 1.0  # Maturity/time period (in years)
S0 = 100  # Current Stock Price
```

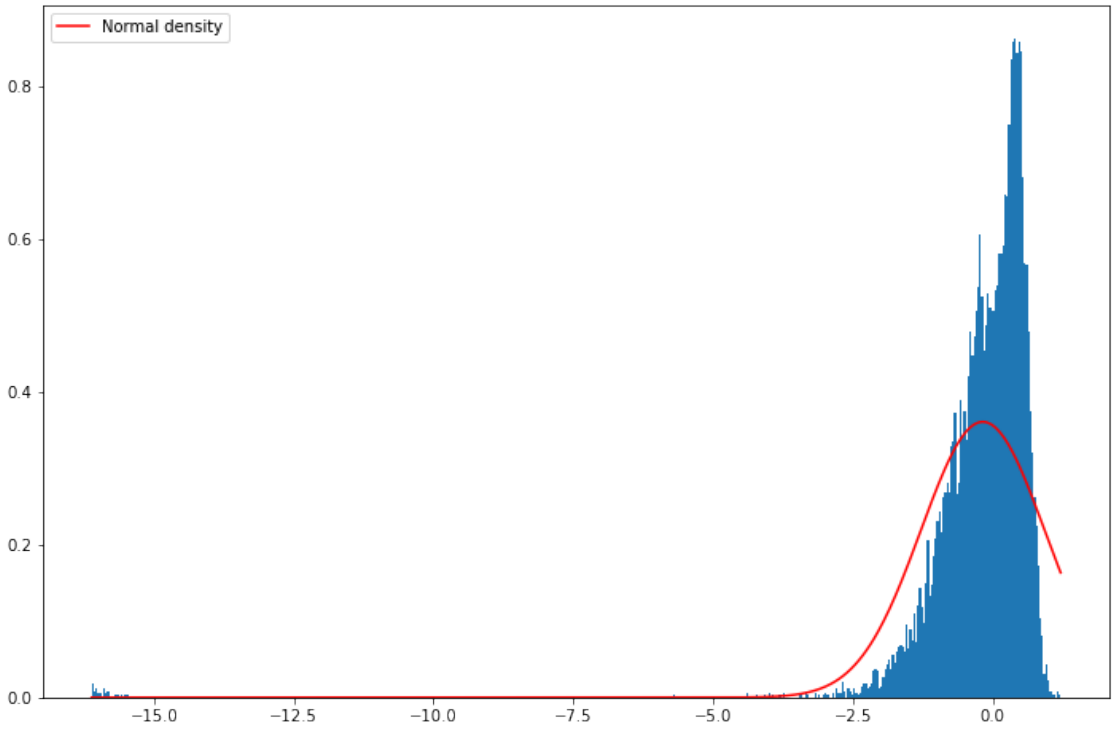
```
Ite = 10000  # Number of simulations (paths)
M = 50  # Number of steps
dt = T / M  # Time-step

SM = np.zeros((M + 1, Ite))
SM[0] = S0

# rj
rj = lamb * (np.exp(mu + 0.5 * delta**2) - 1)

# Random numbers
z1 = np.random.standard_normal((M + 1, Ite))
z2 = np.random.standard_normal((M + 1, Ite))
y = np.random.poisson(lamb * dt, (M + 1, Ite))
```

[23]:
```
for t in range(1, M + 1):
    SM[t] = SM[t - 1] * (
        np.exp((r - rj - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z1[t])
        + (np.exp(mu + delta * z2[t]) - 1) * y[t]
    )
    SM[t] = np.maximum(
        SM[t], 0.00001
    )  # To ensure that the price never goes below zero!
```

[24]:
```
plt.figure(figsize=(10, 8))
plt.plot(SM[:, 100:150])
plt.title("Merton '76 Stock Price Paths")
plt.xlabel("Time Step")
plt.ylabel("Underlying Price")
plt.show()
```



[25]:
```
retSM = np.log(SM[-1, :] / S0)
x = np.linspace(retSM.min(), retSM.max(), 500)

plt.figure(figsize=(12, 8))
plt.hist(retSM, density=True, bins=500)
plt.plot(
    x, ss.norm.pdf(x, retSM.mean(), retSM.std()), color="r", label="Normal␣
 ↪density"
)
plt.legend()
plt.show()
```



If you compare this density graph to the one we obtained in the case of $\lambda = 0.75$, you will observe that the main difference occurs in the extreme left tail of the distribution. This makes sense, since by increasing λ we are effectively influencing the probability that the stock price experiences more negative returns (jumps).

What if we decrease λ?

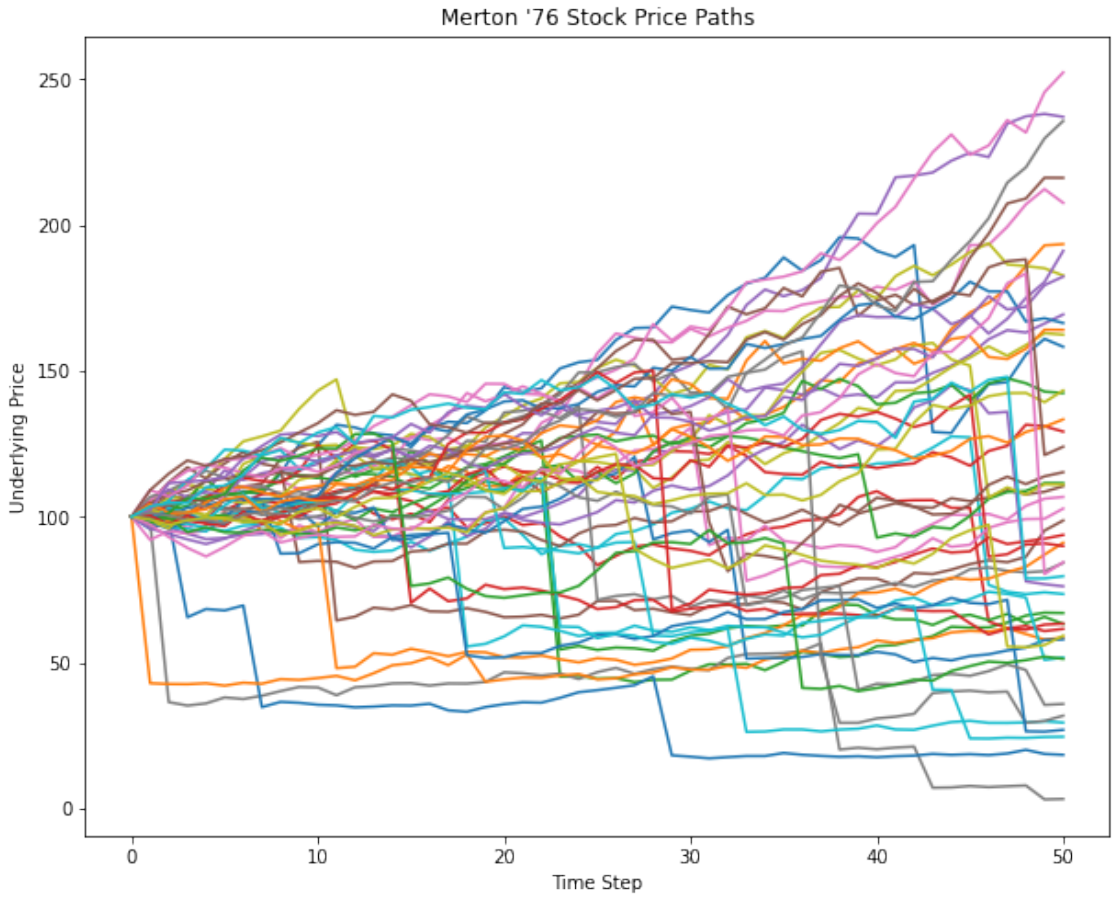
1.5.2 4.2. Changing μ_j

In the previous case, we observe how the distribution's negative skewness increases as a consequence of a higher λ. But why the left tail? Essentially, because we have a $\mu_j < 0$. That is, the average jump size is negative. This means that every time the stock price experiences a jump, there is a very high likelihood that this is a negative jump.

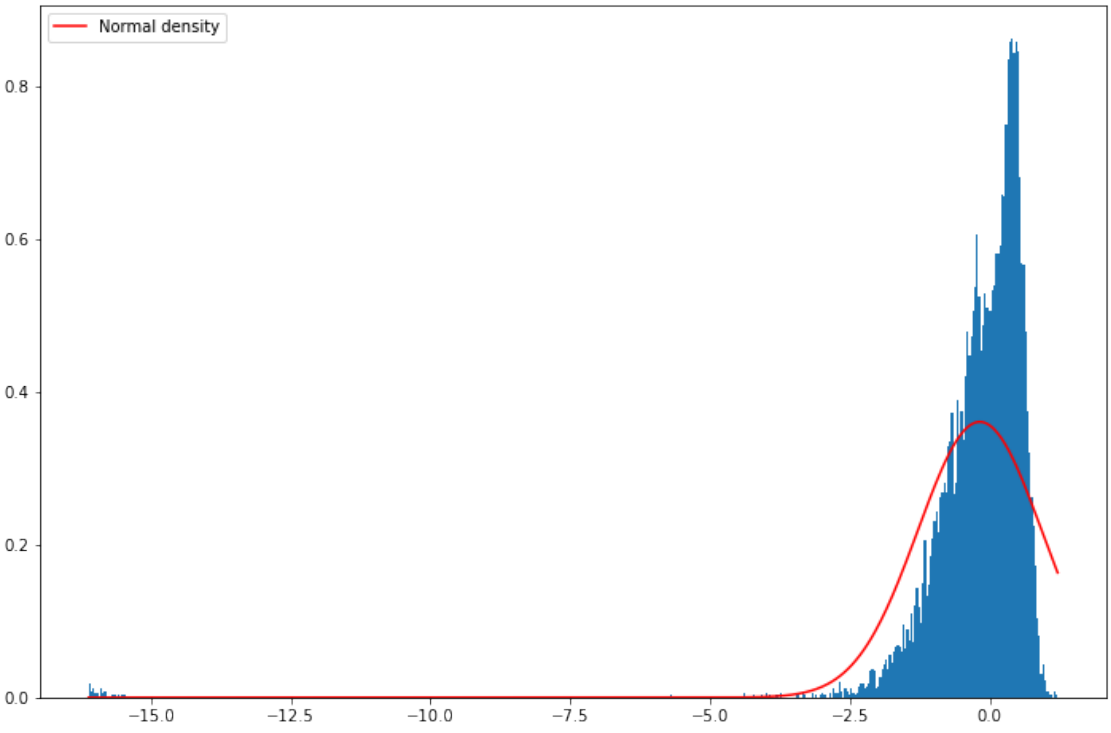
What if we set $\mu_j > 0$?

[26]:
```
for t in range(1, M + 1):
    SM[t] = SM[t - 1] * (
        np.exp((r - rj - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z1[t])
        + (np.exp(mu + delta * z2[t]) - 1) * y[t]
    )
    SM[t] = np.maximum(
        SM[t], 0.00001
    )  # To ensure that the price never goes below zero!
```

[27]:
```
plt.figure(figsize=(10, 8))
plt.plot(SM[:, 0:50])
plt.title("Merton '76 Stock Price Paths")
plt.xlabel("Time Step")
plt.ylabel("Underlying Price")
plt.show()
```



[28]:
```
retSM = np.log(SM[-1, :] / S0)
x = np.linspace(retSM.min(), retSM.max(), 500)

plt.figure(figsize=(12, 8))
plt.hist(retSM, density=True, bins=500)
plt.plot(
    x, ss.norm.pdf(x, retSM.mean(), retSM.std()), color="r", label="Normal␣
 ↪density"
)
plt.legend()
plt.show()
```



Now, as you see, the distribution of returns has skewness (heavier right tail) than normal.