Merton Jump Diffusion Process for Stock Prices Modelling

Submitted in partial fulfillment of the requirements of
MATH F424 Applied Stochastic Process

By

Aditya Bhardwaj 2018A2PS0599P

Anirudh Singh 2019A7PS0107P

Dhananjay Ojha 2018B5A20132P

Under the supervision of:
Dr. Anirudh Singh Rana

BITS Pilani, Pilani Campus

April, 2022

Abstract

In this paper, we compare two stock price models : Black Scholes model and Merton Jump Diffusion model. We first derive the underlying ideas behind both the models, then present a method to simulate them. We finally compare the performance of these models on multiple stocks traded in the Indian Stock Market. Our experiments found that the Merton Jump Diffusion Model (MJD) provides a much better approximation to the empirical distribution of the returns, compared to the Black Scholes Model (BSM). They also indicate that the difference in performance is even more evident in newer stocks with higher volatility, which are thus better modeled using MJD.

Contents

1 Introduction

Introduced in 1970s, Black-Scholes (BS) Model for option pricing was a groundbreaking work, due to which it won Nobel Prize in Economics. But, it was based on one key assumption i.e. stock prices follow Geometric Brownian Motion (GBM). Soon practitioners discovered that these assumptions although largely true, are not exactly correct. Since then the model has been generalized in many directions. One of the major shortcomings of BS model is that when some major incident occurs, then public reaction to the new information often causes jumps in stock prices.

Consequently, this causes the real life distribution of stock returns to be sometimes significantly different from the normal distribution as theoretically predicted by BS model. This creates financial risk for investors worldwide, who are heavily involved in Stock Markets. To mitigate this risk Merton Jump Diffusion (MJD) model was introduced by researchers. MJD is a generalized version of BS model. It allows for poisson jumps with, with exponential waiting times and follows GBM meanwhile. In the further sections it is investigated that whether jump diffusion model is a better fit for stocks than Black-Scholes.

2 Black-Scholes Model

Black Scholes Model assumes that stock prices follow a Geometric Brownian Motion. GBM contains two component i.e. drift and diffusion term. According to GBM, long term trend in stock price are caused by drift term and short term random movements are due to diffusion term. Stock prices follow the SDE

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{1}$$

where, W_t is a Wiener process or Brownian motion, and μ ('Average Drift'), σ ('Volatility') are constants. The former is used to model deterministic trends, while the latter term is often used to model a set of unpredictable events occurring during this motion.For an arbitrary initial value S_0 the above SDE has the analytic solution (under Itô's interpretation),

$$S_T = S_0 \exp\left[\left(\mu - \frac{\sigma^2}{2}\right) T + \sigma \int_0^T dW_t\right] \tag{2}$$

Solution of the above SDE requires Itô calculus. Applying Itô's formula leads to,

$$d(\ln S_t) = (\ln S_t)' \, dS_t + \frac{1}{2} (\ln S_t)'' \, dS_t dS_t = \frac{dS_t}{S_t} - \frac{1}{2}\frac{1}{S_t^2} dS_t dS_t \tag{3}$$

where $dS_t dS_t$ is simply quadratic variation of the SDE.

$$dS_t dS_t = \sigma^2 S_t^2 dW_t dW_t + 2\sigma S_t^2 \mu dW_t dt + \mu^2 S_t^2 dt^2$$

And since,

$$dW_t dW_t = dt \qquad dW_t dt = 0 \qquad dtdt = 0$$

Eq. 3 simplifies to

$$d(\ln S_t) = \left(\mu - \frac{\sigma^2}{2}\right) dt + \sigma dW_t \tag{4}$$

Finally

$$R_{\Delta T} = \ln \frac{S_T}{S_0} = \left(\mu - \frac{\sigma^2}{2}\right) \Delta T + \sigma \int_0^T dW_t \tag{5}$$

3 Merton Jump Diffusion Model

The Jump Diffusion Model is a combination of a Geometric Brownian Motion to model the general trend in the stock prices, and a Compound Poisson process to model the sudden jumps/falls. According to MJD stock prices are governed by the following SDE

$$dS_t = \mu_d S_t dt + \sigma_d S_t dW_t + S_t dJ_t \tag{6}$$

where W_t is brownian motion and $J_t = \sum_{k=1}^{N_t} Y_k$ is compound poisson process as desscribed by Eq. (24).

Solving the above SDE gives the stock prices as:

$$S_T = S_0 \exp\left[\left(\mu_d - \frac{\sigma_d^2}{2}\right) T + \sigma_d \int_0^T dW_t + \int_0^T dJ_t\right] \tag{7}$$

Where S_0 is the stock price at the beginning of the period and S_T is the stock price at time T. Here $\mu_d,\ \sigma_d$ are the diffusion drift and volatility, respectively, and Y_k is the k^{th} jump intensity.

$$J_t = \left\{\sum_{k=1}^{N_t} Y_k\right\}_{t \geqslant 0}$$

is the compound poisson process with normally distributed jumps sampled from $\mathcal{N}(\mu_j, \sigma_j^2)$.

Taking natural log on both sides of Eq. 7,

$$\ln S_T = \ln S_0 + \left(\mu_d - \frac{\sigma_d^2}{2}\right) T + \sigma_d \int_0^T dW_t + \int_0^T dJ_t \tag{8}$$

Therefore, log returns of the stock price as defined in 7 is defined as:

$$R_{\Delta T} = \ln \frac{S_T}{S_0} = \left(\mu_d - \frac{\sigma_d^2}{2}\right) \Delta T + +\sigma_d \int_0^T dW_t + \int_0^T dJ_t \tag{9}$$

such that,

$$S_T = S_0 e^{R_{\Delta T}} \tag{10}$$

4 Estimation of Model Parameters

Maximum Likelihood Estimation (MLE) method is used to estimate the model parameters. The distribution of log returns following MJD model is given by

$$f_{R_{\Delta t}}(x) = \sum_{k=0}^{\infty} p_k(\lambda \Delta t)\ \varphi(x|(\mu_d - \frac{\sigma_d^2}{2})\Delta t + \mu_j k, \sigma_d^2 \Delta t + \sigma_j^2 k) \tag{11}$$

where, $p_k(\lambda \Delta t) = \mathbb{P}\{\Delta N = k\} = \frac{(\lambda \Delta t)^k}{k!} e^{-\lambda \Delta t}$ and φ is gaussian pdf. Eq (11) is used to form likelihood function.

$$L(\theta; x) = \prod_{i=0}^{n} f_{R_{\Delta t}}(x_i) \tag{12}$$

Taking log on both sides gives,

$$-\ln L(\theta; x) = \sum_{i=0}^{n} -\ln f_{R_{\Delta t}}(x_i) \tag{13}$$

In above Eq (12) & Eq (13) $f_{R_{\Delta t}}(x_i)$ is calculated using log returns data. It is often convenient to minimize log likelihood function. Any numerical solver can be used to minimize the above function, with the constraint $\lambda \geq 0$.

5 Simulation

5.1 Poisson Process

To simulate a poisson process, the fact that the waiting time between adjacent poisson events is exponentially distributed is used. Thus, to get the occurence time of $(k+1)^{th}$ poisson event, a randomly sampled τ from the exponential distribution is taken and added to the time of $(k)^{th}$ poisson event.

First, a random sample from exponential doistribution with parameter λ is taken. This is used as the occurence time (τ_1) for first poisson event. Next another sample (τ_2) is taken from exponential distribution and added to τ_1. This is occurence time for second poisson event. This process is repeated.

Formally,

$$T_n = \sum_{k=1}^{n} \tau_i \tag{14}$$

where τ_i, $\forall\ i$ are exponentially distributed and T_n is the time of occurence of n^{th} poisson event.

5.2 Compound Poisson Process

Compound poisson process are simulated just like poisson process, except that now increment at each jump is Y_k where,

$$Y_k \sim \mathcal{N}(\mu_j, \sigma_j^2) \tag{15}$$

At each instant Y_k is sampled from $\mathcal{N}(\mu_j, \sigma_j^2)$ along with τ_k from exponential Distribution. J_t is then calculated as,

$$J_t = \sum_{k=1}^{N_t} Y_k \tag{16}$$

5.3 Geometric Brownian Motion

For given parameters μ and σ Geometric Brownian Motion is simulated by randomly sampling a standard normal variable ϵ, such that $\epsilon \sim \mathcal{N}(0, 1)$. Then, stock price at next instant are calculated using previous price using the following equation

$$S_{t+\Delta t} = S_t \exp\left[\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\epsilon\Delta t\right] \tag{17}$$

Here μ is average rate of return and σ is average volatility.

5.4 Jump Diffusion Process

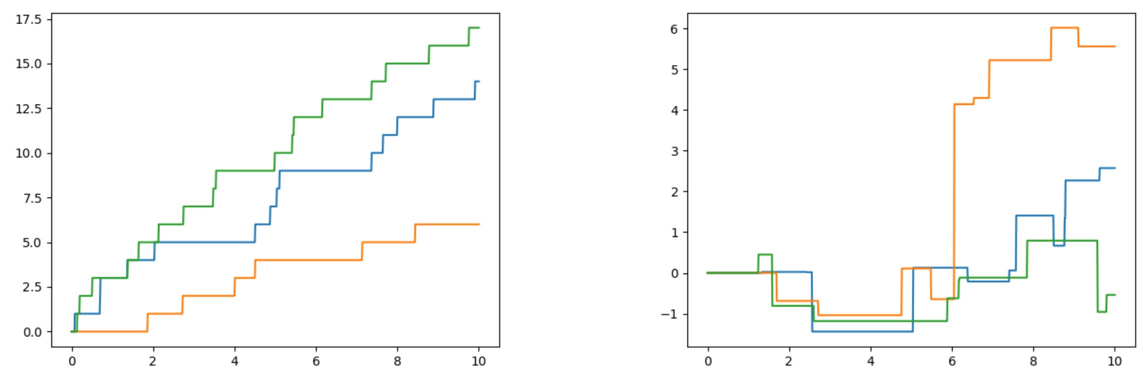
To simulate a MJD process, first Geometric Brownian motion and a Compound Poisson Process are simulated independently. Then the log returns of the two process are added to calculate the total return at each point of time (9) The stock price is then computed from the log returns by using the Eq. (10)

$$S_T = S_0 e^{R_{\Delta T}} \tag{18}$$

6 Results

6.1 Simulation

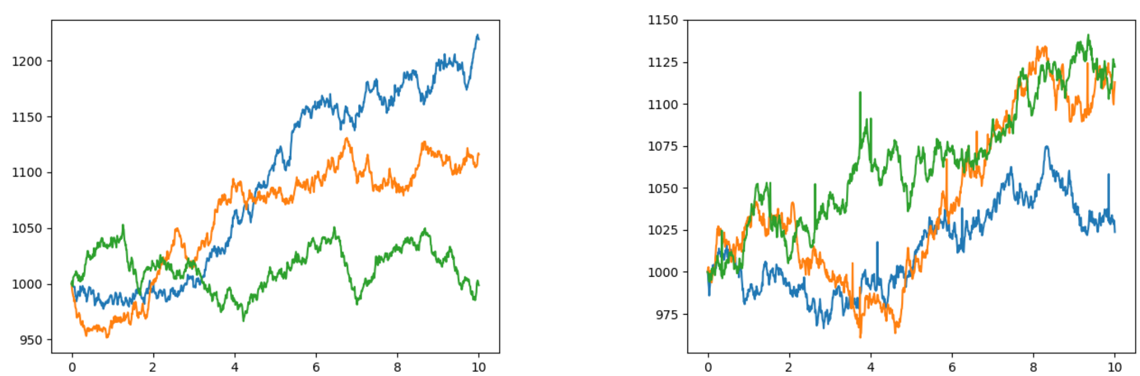
Following results were obtained by running Monte Carlo Simulations of the different process described earlier



(a) Poisson Process

(b) Compound Poisson Process

Figure 1: Monte Carlo Simulations of Poisson and Compound Poisson Process.



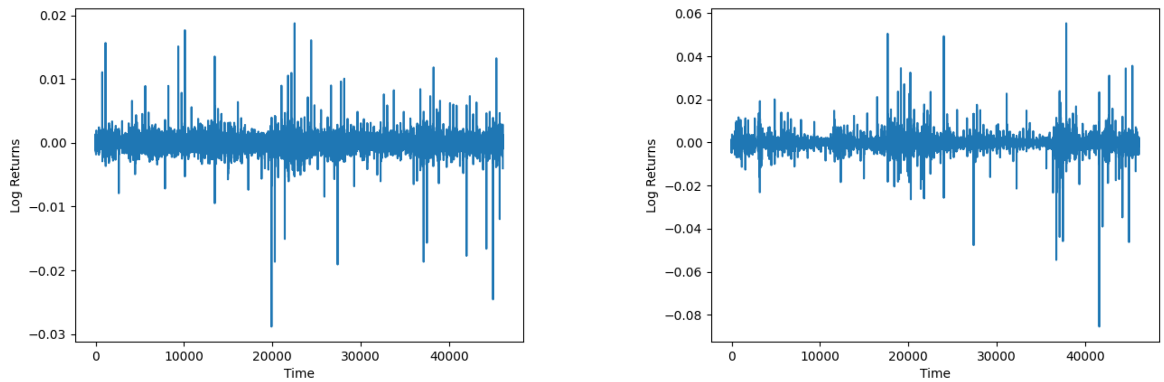
(a) Geometric Brownian Motion

(b) Jump DIffusion Process

Figure 2: Monte Carlo Simulations of Geometric Brownian Motion and Jump Diffusion Process.

6.2 Parameter Estimation

Model simulations can be run after estimating parameters by using the method (4) mentioned above. To calibrate model, first log returns should be found. Below are graph of log returns of two stocks 'Reliance' and 'Zomato'.



(a) Reliance log returns

(b) Zomato log returns

Figure 3: Log Returns calculated at 1 minute frequency from Sep, 2021 to Feb, 2022.

Open-Source python package 'Scipy' was used to minimize the above described log likelihood function (13). Results obtained by applying Maximum Likelihood Estimation method to obtain parameters for some stocks are presented in the following table(1).

Stock	λ	μ_d	σ_d	μ_j	σ_j	$-\ln L$
Reliance	1.30×10^{-9}	9×10^{-3}	0.131	3.358	1.265	-323.90
ONGC	1.55×10^{-10}	1.36×10^{-2}	0.150	-7.64 $\times 10^{-2}$	2.87×10^{-3}	-291.96
Zomato	0.037	0.0149	0.177	-0.0701	0.2428	-236.22
Nykaa	0.480	0.69×10^{-2}	0.1393	-0.91×10^{-2}	0.150	-152.48

Table 1: Estimated parameters by MLE for different stocks for $\Delta t = 1$ day

An interesting obervation from the above Table (1) is that, stocks of large-cap companies such as Reliance and ONGC have very low values of jump parameter λ, as compared to newly listed startups such as Zomato and Nykaa. It means that on average, there are more jumps in stock price of newly listed companies, which tend to be more volatile than large-cap companies. This is inline with the general perception that, public reacts sharply to any new information or news that comes out related to new companies such as Zomato.

According to Black-Scholes,

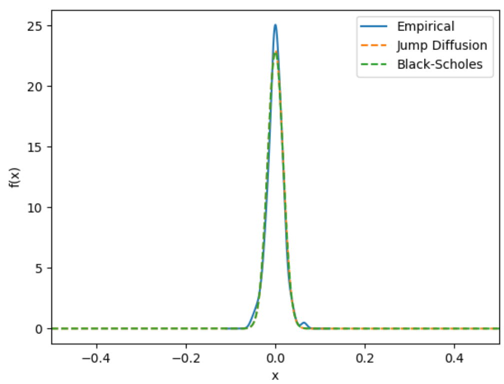
$$R_{\Delta T} = \ln \frac{S_T}{S_0} = (\mu - \frac{\sigma^2}{2})\Delta T + \sigma \int_0^T dW_t \tag{19}$$

Therefore, log returns follows normal distribution with mean and variance as following,

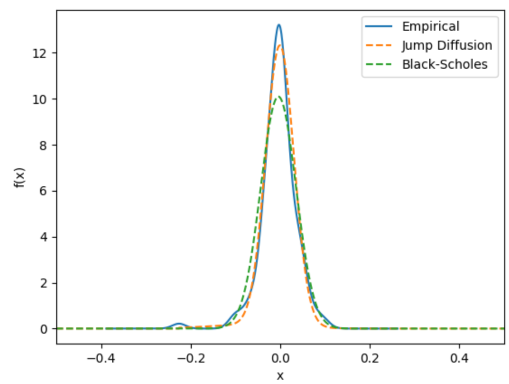
$$\mathbb{E}\left[R_{\Delta T}\right] = (\mu - \frac{\sigma^2}{2})\Delta T \tag{20}$$

$$\operatorname{Var}\left[R_{\Delta T}\right]=\sigma^{2} \Delta T \tag{21}$$

Using the Eq (11) and above equations, the empirical distributions of stock returns can be directly compared with those predicted by BS and MJD model.



(a) Reliance



(b) Zomato

Figure 4: Daily log returns distribution

In the above figures (4), it can be seen that Jump Diffusion model gives a much better approximation to empirical distribution of returns.

7 Conclusion

To conclude, we have discussed and compare the Black-Scholes model and the Merton Jump-Diffusion model in the indian context. We try to find whether the MJD model is significantly more suitable for empirical stock prices. We apply the two models to simulate the prices of different stocks traded in the Indian stock market, and present our findings here. The parameters for the two models that best fit the empirical data were found using Maximum Likelihood Estimation. From our experiments, we found that newly listed stocks have higher volatility compared to large-cap companies, with much more frequent price jumps. Since the MJD model is able to model fluctuations caused by jumps/falls, it performs much better than a simble SBM. Based on visual evidence, we found that the MJD model is better able to predict the future movement of stock prices.

Appendix

A Mathematical Preliminaries

A.1 Poisson Process

A Poisson Process $\boldsymbol{N} = \{N_t : t \geqslant 0\}$ is a counting process, which is used to model occurence of random events such that their interarrival times are exponentially distributed with parametrer λ. The number of events between any time interval $(t,\ t + \Delta t)$ follows poisson distribution, as given in Eq. (22)

$$\mathbb{P}\{N = k\} = \frac{(\lambda \Delta t)^k}{k!} e^{-\lambda \Delta t} \tag{22}$$

with,

$$\mathbb{E}\left[N\right] = \lambda \Delta t \tag{23}$$

Poisson process is used to model a wide variety of process for e.g. such as customer arrival at a store, phone calls at a switchboard, or earthquakes etc.

A.2 Compound Poisson Process

Compound Poisson Process $\boldsymbol{J} = \{J_t : t \geqslant 0\}$ are a generalilzation of poisson process. For these process the interarrival time between random events is still exponentially distributed, but the size of jump is not 1, rather it is randomly distributed according to some law F. Commonly, the law F is assumed to be Normal Distribution.

At any instant, the value of process is given by the sum of the jump size (Y_k) till that instant. such that,

$$J_t = \sum_{k=1}^{N_t} Y_k \tag{24}$$

where, N_t is number of poisson events till time t. Therefore, a compound Poisson process is a real-valued right-continuous process $(Z_t : t \geqslant 0)$ with the following properties.

1. Finitely many jumps: for all $\omega \in \Omega$, sampled path $t \mapsto J_t(\omega)$ has finitely many jumps in finite intervals.
2. Independent increments: for all $t,\ s \geqslant 0$; $J_{t+s} - J_t$ is independent of past $\{J_u : u \leq t\}$,
3. Stationary increments: for all $t,\ s \geqslant 0$, distribution of $J_{t+s} - J_t$ depends only on s and not on t.

A.3 Brownian Motion

A stochastic process $\boldsymbol{W} = \{W_t : t \geqslant 0t\}$ is called a standard Brownian Motion if the following properties hold:

1. $W_0 = 0$.
2. Independent increments: for every time point $0 \leqslant t_1 \leqslant t_2 \leqslant \ldots \leqslant t_n$, the increments of W: $W_{t_n} - W_{t_{n-1}}$, $W_{t_{n-1}} - W_{t_{n-2}}$, $\ldots$, $W_{t_1} - W_{t_0}$ are independent random variable.
3. Normal distribution: For every Δt, the increment $\Delta W_t = W_{t+\Delta t} - W_t \sim \mathcal{N}(0, \Delta t)$
4. Almost surely, the function $t \to W_t$ is a continuous function for every t.

B Code

B.1 Monte Carlo

```python
#/usr/bin/env python3

"""
Author:
- Aditya Bhardwaj <adityabhardwaj727@gmail.com>

This code is part of the project Merton Jump Diffusion
Process for Stock Price Modelling. This work was done in
partial fulfilment of the course MATH F424 (pplied Stochastic Process),
Mathematics Department, BITS Pilani, India

Data:
Stock Market Data gathered from Yahoo Finance
More details can be found at the link provided below
https://github.com/SneakyRain/Jump-Diffusion

"""

import bisect
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

class MonteCarlo():
    """
    Monte Carlo class
    """
    def __init__(self, T, num_steps, num_sim) -> None:
        """
        Init MonteCarlo

        Inputs
        ------
        `T`: total time
        `num_steps`: number of total steps
        `num_sim`: number of simulation to run
        """
        self.num_steps = num_steps
        self.T = T
        self.num_sim = num_sim
        self.dt = self.T/self.num_steps

    def poisson_process(self, _lambda):
        """
        Simulate Poisson Process

        Inputs
        ------
        `_lambda`:  mean rate of occurence of random `poisson` jumps

        Outputs
        -------
        `values`: simulated values of poisson process

```

```
        References
        ----------
        1. `Poisson Process`: https://en.wikipedia.org/wiki/Poisson_point_process
        """
        event_times = []
        events = []
        t = count = 0
        while True:
            tau = stats.distributions.expon(scale=1/_lambda)
            t = t + tau.rvs()
            count = count + 1
            if t <= self.T:
                event_times.append(t)
                events.append(count)
            else:
                break

        self.tgrid = np.linspace(0, self.T, self.num_steps+1)
        values = []
        for t_i in self.tgrid:
            id = bisect.bisect(event_times, t_i)
            if id == 0:
                v = 0
            else:
                v = events[id-1]
            values.append(v)
        values = np.array(values)
        return values

    def compound_poisson_process(self, _lambda, f):
        """
        Simulate Compound Poisson Process

        Inputs
        ------
        `_lambda`:  mean rate of occurence of random `poisson` jumps
        `f`: law for jump intensity

        Outputs
        -------
        `values`: simulated values of compound poisson process

        References
        ----------
        1. `Compound Poisson Process`: https://en.wikipedia.org/wiki/Compound_Poisson_process
        """
        event_times = []
        events = []
        t = count = 0
        while True:
            tau = stats.distributions.expon(scale=1/_lambda)
            t = t + tau.rvs()
            jump = f.rvs()
            count = count + jump
            if t <= self.T:
                event_times.append(t)
                events.append(count)
            else:
```

```python
                break

        self.tgrid = np.linspace(0, self.T, self.num_steps+1)
        values = []
        for t_i in self.tgrid:
            id = bisect.bisect(event_times, t_i)
            if id == 0:
                v = 0
            else:
                v = events[id-1]
            values.append(v)
        values = np.array(values)
        return values

    def geometric_brownian_motion(self, mu, sigma, S0):
        """
        Simulate Geoetric Brownian Motion

        Inputs
        ------
        `mu`: mean of drift
        `sigma`: std of diffusion
        `S0`: initial stock price

        Outputs
        -------
        `values`: simulated values of brownian motion

        References
        ----------
        1. Geometric Brownian Motion: https://en.wikipedia.org/wiki/
Geometric_Brownian_motion
        """
        self.tgrid  = np.linspace(0, self.T, self.num_steps+1)
        values = np.exp(
            (mu - 0.5*(sigma**2)) * self.dt
            + sigma * np.random.normal(0, np.sqrt(self.dt), size=(self.num_sim,
self.num_steps)).T
        )
        values = np.vstack([np.ones(self.num_sim), values])
        values = S0 * values.cumprod(axis=0)
        return values

    def get_gbm_log_increments(self, mu, sigma):
        """
        Simulate Geoetric Brownian Motion log-returns

        Inputs
        ------
        `mu`: mean of drift
        `sigma`: std of diffusion

        Outputs
        -------
        `ts`: time steps
        `ys`: simulated values of brownian motion

        References
        ----------
```

```
        1. Geometric Brownian Motion: https://en.wikipedia.org/wiki/Geometric_Brownian_motion
        """
        # simulation using numpy arrays for the geometiric brownian motion
        ys = [0]
        inc = 0

        for i in range(self.num_steps):
            inc += (mu - sigma ** 2 / 2) * self.dt + sigma * np.random.normal(0, np.sqrt(self.dt))
            ys.append(inc)

        ts = np.linspace(0, self.T, self.num_steps)
        return ts, ys


    def jump_process(self, _lambda, mu_d, sig_d, mu_j, sig_j, S0):
        """
        Simulate Jump Diffusion Process

        Inputs
        ------
        `_lambda`:  mean rate of occurence of random `poisson` jumps
        `mu_d`: mean of drift
        `sig_d`: standard deviation of drift
        `mu_j`: mean of normally distributed jumps
        `sig_j`: standard deviation of normally distributed jumps
        `S0`: initial stock price

        Outputs
        -------
        `values`: simulated values of jump diffusion

        References
        ----------
        1. `Jump Diffusion Process`: https://en.wikipedia.org/wiki/Jump_diffusion
        """

        self.tgrid = np.linspace(0, self.T, self.num_steps+1)
        t_gbm, y_gbm = self.get_gbm_log_increments(mu_d, sig_d)

        f = stats.distributions.norm(loc=mu_j, scale=sig_j)
        compound = self.compound_poisson_process(_lambda, f)
        compound = np.array(compound)
        # print(compound[1:])
        # print(compound[:-1])
        increments = compound[1:] - compound[:-1]
        increments = np.insert(increments, 0, 0)

        log_returns = []
        for i, _ in enumerate(self.tgrid):
            r = y_gbm[i] + increments[i]
            log_returns.append(r)

        values = S0 * np.exp(log_returns)
        return values

if __name__ == "__main__":
    mc = MonteCarlo(10, 1000, 1)
```

```python
    f = stats.distributions.norm(0, 1)
    val = mc.poisson_process(1)
    plt.plot(mc.tgrid, val)
    val = mc.poisson_process(1)
    plt.plot(mc.tgrid, val)
    val = mc.poisson_process(1)
    plt.plot(mc.tgrid, val)
    plt.show()
```

B.2 Parameter Estimation

```python
#/usr/bin/env python3

"""
Author:
- Aditya Bhardwaj <adityabhardwaj727@gmail.com>

This code is part of the project Merton Jump Diffusion
Process for Stock Price Modelling. This work was done in
partial fulfilment of the course MATH F424 (pplied Stochastic Process),
Mathematics Department, BITS Pilani, India

Data:
Stock Market Data gathered from Yahoo Finance
More details can be found at the link provided below
https://github.com/SneakyRain/Jump-Diffusion

"""

import numpy as np
import pandas as pd
import scipy.stats as stats
import scipy.optimize as optimize
import matplotlib.pyplot as plt
global epsilon

epsilon = 1e-10

class ModelCalibration():
    """
    Model Calibration class
    """
    def __init__(self, prices, dt) -> None:
        """
        Init ModelCalibration

        Inputs
        ------
        `prices`: daily stock prices
        `dt`: `1 day` (default) time interval
        """
        self.prices = prices
        self.dt = dt
        self.sampling_freq = "1D"
        self.prices = self.data_preprocessing(self.prices, self.sampling_freq)
        self.returns = self.calculate_log_returns(self.prices.close)

    @staticmethod
    def data_preprocessing(m_data, sampling_freq):
```

```
        """
        Pre process the data before analyzing it. This function changes the sampling frequency of ohlc data to given
        sampling frequency.

        Inputs
        ------
        `m_data`: raw market data in ohlc (open-high-low-close) format
        `sampling_freq`: sampling frequency

        Outputs
        -------
        `m_data`: preprocessed market data
        """
        conversion = {'open': 'first',
                      'high': 'max',
                      'low': 'min',
                      'close': 'last',
                      'volume': 'sum'
        }
        resampled_m_data = m_data.resample(sampling_freq).agg(conversion).dropna()
        return resampled_m_data

    @staticmethod
    def calculate_log_returns(prices):
        """
        Calculate log returns of prices

        Inputs
        ------
        `prices`: variable containing prices

        Outputs
        -------
        `lr`: log return of prices
        """
        lr = np.log(prices/prices.shift(1)).dropna()
        return lr

    def jump_pdf(self, _lambda, mu_d, sig_d, mu_j, sig_j):
        """
        Returns pdf function for returns of stock following `Merton Jump Diffusion Model`

        Inputs
        ------
        `_lambda`:  mean rate of occurence of random `poisson` jumps
        `mu_d`: mean of drift
        `sig_d`: standard deviation of drift
        `mu_j`: mean of normally distributed jumps
        `sig_j`: standard deviation of normally distributed jumps

        Outputs
        -------
        `f`: pdf for given jump diffusion characteristics

        References
        ----------
        1. `Poisson Process`: https://en.wikipedia.org/wiki/Poisson_point_process
```

```
        2. `Jump Diffusion Process`: https://en.wikipedia.org/wiki/Jump_diffusion
        """
        def f(x):
            """
            Calculates pdf for given x

            Inputs
            ------
            `x`: x

            Outputs
            -------
            `ans`: pdf value for given input (`x`)
            """
            k = ans = 0
            increment = 1
            while increment >  epsilon:
                pk = stats.distributions.poisson.pmf(k, _lambda*self.dt)
                mean = (mu_d - (sig_d**2)/2)*self.dt + mu_j*k
                std = (sig_d**2)*self.dt + (sig_j**2)*k
                phi = stats.distributions.norm.pdf(x, mean, std)
                increment = pk * phi
                ans = ans + increment
                k = k+1
            if ans == 0:
                ans = epsilon
            return ans
        return f

    def log_likelihood(self, args):
        """
        Calculate negative log likelihood for given args and data

        Inputs
        ------
        `args`: a `List` containing the following parameters
                `_lambda`:  mean rate of occurence of random `poisson` jumps
                `mu_d`: mean of drift
                `sig_d`: standard deviation of drift
                `mu_j`: mean of normally distributed jumps
                `sig_j`: standard deviation of normally distributed jumps

        Outputs
        -------
        `sum`: negative log likelyhood of the jump pdf of daily log returns
        """
        _lambda = args[0]
        mu_d = args[1]
        sig_d = args[2]
        mu_j = args[3]
        sig_j = args[4]
        sum = 0
        self.f = self.jump_pdf(_lambda, mu_d, sig_d, mu_j, sig_j)
        for r in self.returns:
            sum = sum + np.math.log(self.f(r))
        return -sum

if __name__ == "__main__":
    prices = pd.read_csv('data/zomato.csv', parse_dates=True, index_col="timestamp
```

```
    ")
    dt = 1
    calibration = ModelCalibration(prices, dt)
    l = calibration.calculate_log_returns(prices.close)
    l = l.reset_index(drop=True)
    plt.plot(l)
    plt.ylabel("Log Returns")
    plt.xlabel("Time")
    plt.show()
    x0 = [1, 1, 1, 1, 1]

    calibration.cons = [
        {'type': 'ineq', 'fun': lambda x: x[0]},
        {'type': 'ineq', 'fun': lambda x: x[2]},
        {'type': 'ineq', 'fun': lambda x: x[4]}
    ]
    calibration.res = optimize.minimize(calibration.log_likelihood, x0,
    constraints=calibration.cons)
    print(calibration.res.x)
    print(calibration.res.message)
    print(calibration.log_likelihood(calibration.res.x))
    print(calibration.returns.describe())
```

B.3 Comparison between Models

```
#/usr/bin/env python3

"""
Author:
- Aditya Bhardwaj <adityabhardwaj727@gmail.com>

This code is part of the project Merton Jump Diffusion
Process for Stock Price Modelling. This work was done in
partial fulfilment of the course MATH F424 (pplied Stochastic Process),
Mathematics Department, BITS Pilani, India

Data:
Stock Market Data gathered from Yahoo Finance
More details can be found at the link provided below
https://github.com/SneakyRain/Jump-Diffusion

"""

import numpy as np
import pandas as pd
import scipy.stats as stats
import scipy.optimize as optimize
import matplotlib.pyplot as plt
from parameters import ModelCalibration

if __name__ == "__main__":
    prices = pd.read_csv('data/infy.csv', parse_dates=True, index_col="timestamp")
    dt = 1
    mc = ModelCalibration(prices, dt)

    x0 = [1, 1, 1, 1, 1]

    mc.cons = [
        {'type': 'ineq', 'fun': lambda x: x[0]},
```

```python
        {'type': 'ineq', 'fun': lambda x: x[2]},
        {'type': 'ineq', 'fun': lambda x: x[4]}
    ]
    mc.res = optimize.minimize(mc.log_likelihood, x0, constraints=mc.cons)
    res = mc.res.x
    f = mc.jump_pdf(res[0], res[1], res[2], res[3], res[4])

    x = np.linspace(-1, 1, 10000)
    lr = mc.calculate_log_returns(prices.close)

    mu = mc.returns.mean()
    sig = mc.returns.std()
    bs = stats.distributions.norm.pdf(x, mu, sig)

    mjd = [f(i) for i in x]

    mc.returns.plot.kde(label="Empirical")
    plt.plot(x, mjd, "--", label="Jump Diffusion")
    plt.plot(x, bs, "--", label="Black-Scholes")
    plt.xlim(-0.5, 0.5)
    plt.legend()
    # plt.title("Log Returns Distribution")
    plt.ylabel("f(x)")
    plt.xlabel("x")
    plt.show()
```