week11-02–monte-carlo-simulation-JFTE

March 31, 2025

1 George McNinch Math 87 - Spring 2025

2 Week 11

2.1 # Monte-Carlo simulation – Jane's Fish Tank Emporium

3 A modeling application of Monte-Carlo methods: fish tanks!

In this notebook we are going to discuss a modeling example.

Suppose that you have been promoted to inventory manager at **Jane's Fish Tank Emporium** (JFTE).

- JFTE sells 150 gallon fish tanks that are quite bulky. In particular, it is best to not keep more fish tanks in inventory than are needed at any given point in time.
- on average JFTE sells one tank per week.
- JFTE can order new tanks at any point, but must wait 5 days for the arrival of a new tank once it has been ordered.

The task is to design a good strategy for ordering fish tanks.

4 Relevant questions & parameters??

- profit from the sale of a tank?
- cost of storage for an unsold tank in stock?
- what does "on average, one tank is sold per week" really mean??
- what strategies are even possible?

Let's consider some extremal cases first:

- if the profit per tank is large and the storage costs for an in-stock tank relatively small, then a good strategy is to keep a relatively large inventory.
- if the profit per tank is small and the storage costs for an in-stock tank are relatively large, then a good strategy is to keep little-or-no inventory and order as required.

It is difficult to formulate too many generalities without knowing further information.

An important rule of modeling we'd like to follow is this:

Start with a relatively simple model, but build it to allow incremental additions of complexity.

5 Simplifying assumptions

1. Let's assume that "on average, JFTE sells one tank per week" means that on any given day, there is a $\frac{1}{7}$ chance of an interested customer entering the store.
2. If an interested customer arrives but there is no stock, the potential sale is then *lost* (thus our model doesn't acknowledge rainchecks or instructions to a customer to "try next week").
3. The cost of storing a tank is high enough that you only want to store tanks you expect to sell "soon".

These assumptions suggest two strategies, which we want to compare.

Strategy A. Set a *standing order* to have one tank delivered each week.
Strategy B. Order a new tank whenever one is sold – *on-demand ordering*

We are going to use a Monte-Carlo simulation to compare these two strategies.

6 Our simulation

The first step is to simulate arrival of customers. We are going to make a list of N days for our simulation, and for each day we are going to use a random selection to "decide" whether a customer arrives.

For each day, we would like to keep track of various information:

- does a customer arrive? (determined randomly)
- is there a tank in stock? (ordering is determined by our strategy)

So let's create a `python` data structure which keeps track of the required information. We'll just use a `class` named `JFTE` which has instance variables `customers`, `stock`, `sales` etc.

When we construct an instance of the class, we indicate the number of days `N` for our simulation. We create a list corresponding to `days`, and the random number generated "decides" whether or not a customer will arrive on the given day.

We now implement our *strategies* as functions which take as argument an instance of the class `JFTE` and return dictionary of `result values`.

[1]:
```
import numpy as np
import itertools as it

from numpy.random import default_rng
rng = default_rng()
```

[2]:
```
def customer(prob=1./7):
    return rng.choice([1,0],p=[prob,1-prob])


class JFTE():
    def __init__(self,N,prob=1./7):
        self.customers = [customer(prob) for n in range(N)]
```

```
        self.num_days = N
        self.reset()

    def reset(self):
        self.stock = 1
        self.sales = 0
        self.lost_sales = 0
        self.storage_days = 0
        self.max_stock = 1

    def add_stock(self):
        self.stock = self.stock + 1
        if self.stock > self.max_stock:
            self.max_stock = self.stock

    def sale_attempt(self):
        if self.stock > 0:
            self.stock -= 1
            self.sales += 1
        else:
            self.lost_sales += 1

    def result(self):
        return { 'number_days': self.num_days,
                 'weeks': self.num_days/7.0,
                 'sales': self.sales,
                 'lost_sales': self.lost_sales,
                 'storage_days': self.storage_days,
                 'max_stock': self.max_stock
               }
```

The first strategy is to have a standing order made each week on the same day.

[21]:
```
def stand_order(J,dow=6):
    ## dow = arrival day-of-week for standing order; should be in␣
 ↪[0,1,2,3,4,5,6]
    ## we'll assume that the first day of the ``days`` list is dow=0.

    N = J.num_days
    J.reset()

    # loop through the days
    for i in range(N):
        c = J.customers[i]            ## c is 1 if there is a customer on day␣
 ↪i, 0 otherwise

        if dow == np.mod(i,7):        ## add stock on the dow for order arrival
```

```
            J.add_stock()
            J.add_stock()

        if c>0:
            J.sale_attempt()

        J.storage_days += J.stock    ## accumulate total storage costs

    return J.result()
```

The second strategy is to have a order placed as soon as a sale is made.

[4]:
```
np.inf > 0
```

[4]:
```
True
```

[5]:
```
def order_on_demand(J):
    J.reset()
    order_wait = np.inf                          ## order_wait represents␣
 ↪wait-time
                                                 ## until next order arrival
    ## loop through the customers
    for c in J.customers:
        if c>0:                  ## record attempted sale to customer
            J.sale_attempt()

        J.storage_days += J.stock                ## accumulate storage days

        if  J.stock==0 and order_wait == np.inf:  ## reorder if stock is empty␣
 ↪and no current order
            order_wait = 5

        if order_wait == 0:                      ## stock arrives
            J.add_stock()
            order_wait = np.inf

        if order_wait>0:                         ## decrement arrival time for␣
 ↪in-transit orders
            order_wait -= 1

    return J.result()
```

[16]:
```
J = JFTE(2*52*7)   # run for 2 years

stand_result = stand_order(J,dow=6)

demand_result = order_on_demand(J)
```

[17]:
```
stand_result
```

[17]:
```
{'number_days': 728,
 'weeks': 104.0,
 'sales': 94,
 'lost_sales': 0,
 'storage_days': 5864,
 'max_stock': 14}
```

[8]:
```
demand_result
```

[8]:
```
{'number_days': 728,
 'weeks': 104.0,
 'sales': 59,
 'lost_sales': 43,
 'storage_days': 378,
 'max_stock': 1}
```

[22]:
```
import pandas as pd

def make_trials(trial_weeks = 2*52, num_trials = 10):
    return [ JFTE(7*trial_weeks,2.0/7) for _ in range(num_trials) ]

def report_trials(strategy,trials):

    results = [ strategy(t) for t in trials ]

    details = ['weeks', 'sales', 'lost_sales', 'storage_days', 'max_stock']

    sd = {i: [r[i] for r in results ] for i in details}

    return pd.DataFrame(sd)

## make a list of 10 trials. Each trial has length 2 years
ten_trials = make_trials()

# now we can use `ten_trials` as input to `report_trials` and compare
# the results of our ordering strategies on the same data.
```

[23]:
```
stand_results = report_trials(stand_order,ten_trials)
print(stand_results)
```

```
   weeks  sales  lost_sales  storage_days  max_stock
0  104.0    206          22          1774         10
1  104.0    187           0          9618         24
2  104.0    206           8          5387         18
3  104.0    205          10          2528          9
4  104.0    193           0          5066         16
```

```
5  104.0    204          10          4911         20
6  104.0    200           2          5279         14
7  104.0    208          17          3003         13
8  104.0    206           5          3615         12
9  104.0    200          15          3188         12
```

[20]:
```
demand_results = report_trials(order_on_demand, ten_trials)
demand_results
```

[20]:
```
   weeks  sales  lost_sales  storage_days  max_stock
0  104.0     86         128           215          1
1  104.0     84         117           224          1
2  104.0     84         124           224          1
3  104.0     88         127           201          1
4  104.0     85         132           221          1
5  104.0     87         124           211          1
6  104.0     88         119           201          1
7  104.0     85         106           218          1
8  104.0     85         108           219          1
9  104.0     81         108           242          1
```

[12]:
```
stand_results.mean()
```

[12]:
```
weeks           104.0
sales            95.0
lost_sales        6.5
storage_days    4158.0
max_stock         13.2
dtype: float64
```

[13]:
```
demand_results.mean()
```

[13]:
```
weeks           104.0
sales            60.7
lost_sales       40.8
storage_days    365.3
max_stock         1.0
dtype: float64
```

[14]:
```
stand_results.std()
```

[14]:
```
weeks             0.000000
sales             7.731609
lost_sales        6.720615
storage_days    2712.902628
max_stock         6.356099
dtype: float64
```

[15]:
```
demand_results.std()
```

[15]:
```
weeks            0.000000
sales            3.622461
lost_sales       9.531235
storage_days    22.944619
max_stock        0.000000
dtype: float64
```