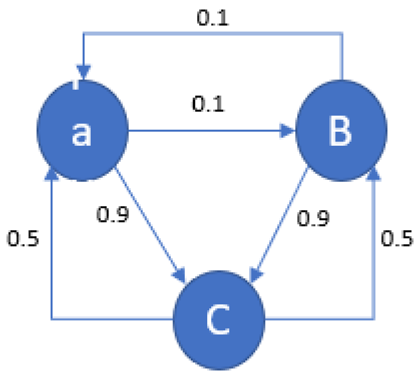
Markov Chain Analysis and Simulation using Python

Solving real-world problems with probabilities

Herman Scheepers
Follow
Nov 20, 2019 · 7 min read

A Markov chain is a discrete-time stochastic process that progresses from one state to another with certain probabilities that can be represented by a graph and state transition matrix P as indicated below:



$$P = \begin{bmatrix} 0 & 0.1 & 0.9 \\ 0.1 & 0.0 & 0.9 \\ 0.5 & 0.5 & 0.0 \end{bmatrix}$$

Such chains, if they are first-order Markov Chains, exhibit the Markov property, being that the next state is only dependent on the current state, and not how it got there:

$s_{i+1} = s_i.P$

$s_{i+1} = s_i.P$

In this post we look at two separate concepts, the one being simulating from a Markov Chain, and the other calculating its stationary distribution. The stationary distribution is the fraction of time that the system spends in each state as the number of samples approaches infinity. If we have *N* states, the stationary distribution is a vector of length N, of which the values sum up to 1, since it's a probability distribution.

We also look at two examples, a simple toy example, as well as a possible real-world scenario analysis problem.

Calculating the Stationary Distribution

Note that in the first implementation below is not a simulation of the state transitions, just a calculation of the stationary distribution.

Let's start with an iterative approach to calculating the distribution. What we are doing is raising the transition matrix to the power of the number of iterations:

$$s_1 = s_0 P, \qquad s_{2=} s_1 P = (s_o P)P = s_0 P^2 \ldots . \qquad s_n = s_0 \, P^n$$

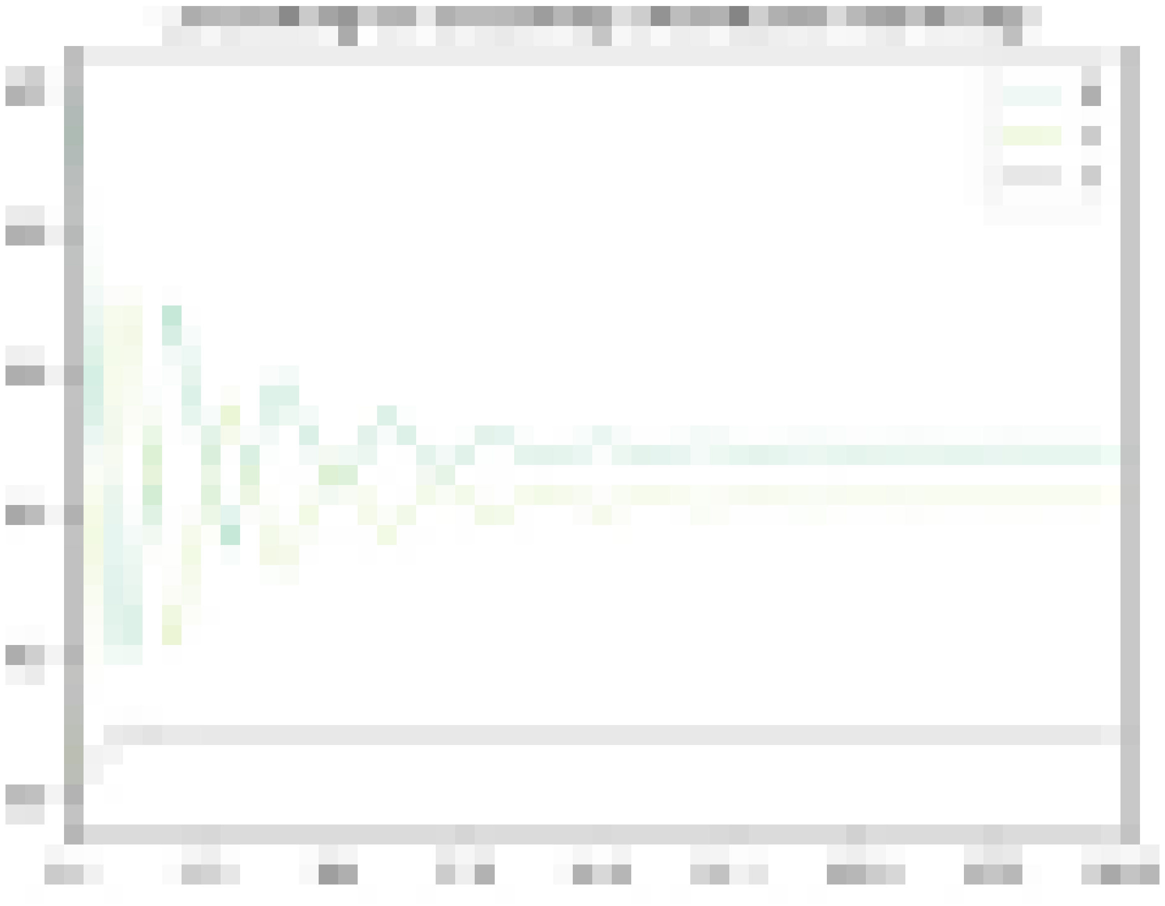
The stationary distribution is usually referred to as π.

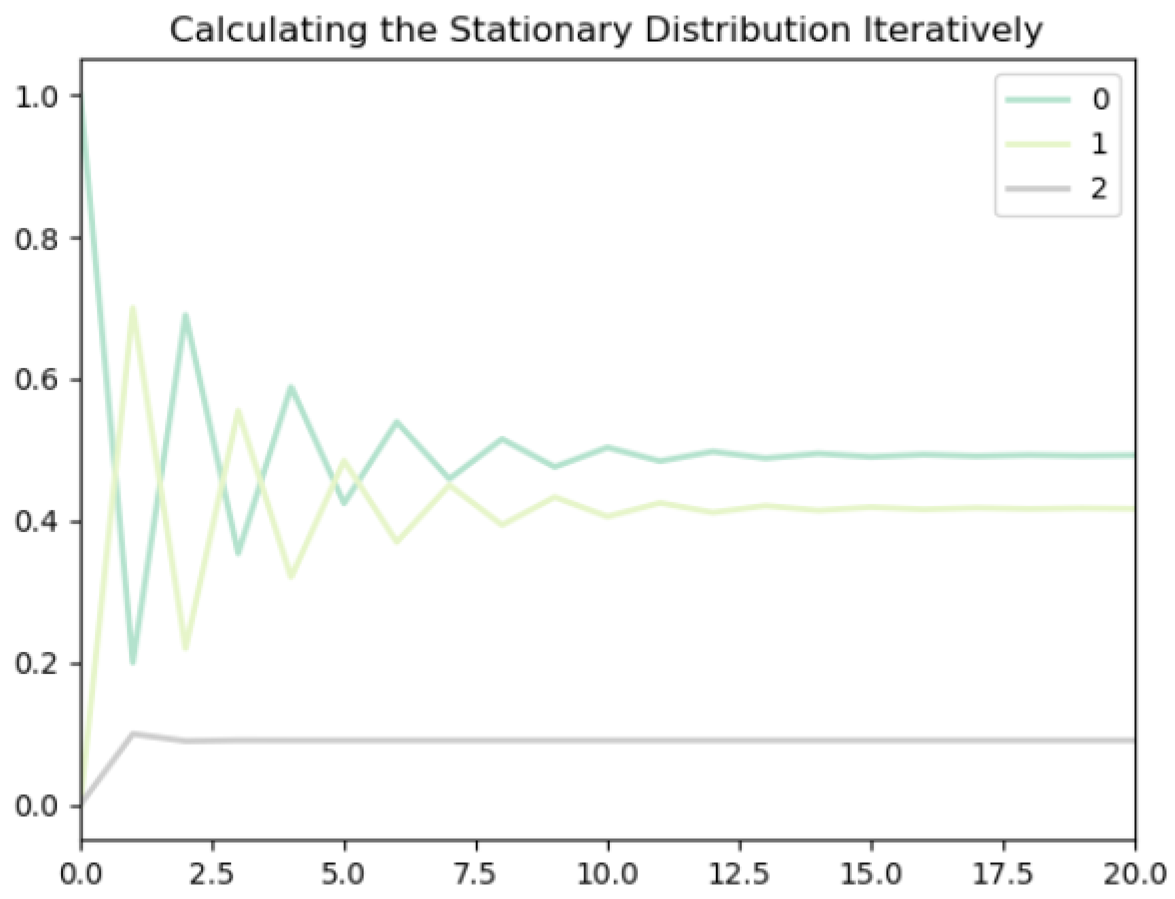
So

$$\pi = s_0 \, P^n \, ; n \rightarrow \infty$$

```
import numpy as np
import pandas as pd
from random import seed
from random import random
import matplotlib.pyplot as plt
P = np.array([[0.2, 0.7, 0.1],
              [0.9, 0.0, 0.1],
              [0.2, 0.8, 0.0]])state=np.array([[1.0, 0.0, 0.0]])
stateHist=state
dfStateHist=pd.DataFrame(state)
distr_hist = [[0,0,0]]for x in range(50):
  state=np.dot(state,P)
  print(state)
  stateHist=np.append(stateHist,state,axis=0)
  dfDistrHist = pd.DataFrame(stateHist)
  dfDistrHist.plot()plt.show()
```

The calculation converges to the stationary distribution quite quickly:





```
Π = [0.49 0.42 0.09]
```

```
Π = [0.49 0.42 0.09]
```

With Pi being the stationary distribution, as described earlier.

This may also be accomplished in this case with a linear algebra solution of a set of over-determined equations:

```
A=np.append(transpose(P)-identity(3),[[1,1,1]],axis=0
b=transpose(np.array([0,0,0,1]))
np.linalg.solve(transpose(A).dot(A), transpose(A).dot(b)
```

Which also returns [0.49, 0.42 , 0.09], the stationary distribution π.

How we got to this calculation is shown below:

It can be shown that a Markov chain is stationary with stationary distribution π if πP=π and πi=1

Where ***i*** is a unit column vector — i.e. the sum of the probabilities must be exactly 1, which may also be expressed as

$$\sum\nolimits_{i\in S} \pi_j = 1$$
$$\sum\nolimits_{i\in S} \pi_j = 1$$

Doing some algebra:

$$\pi\,(P - I) = 0$$

$$\left(P^T - I\right).\,\pi = 0$$

$$\pi\,(P - I) = 0$$

$$\left(P^T - I\right).\,\pi = 0$$

Combining with $\pi \boldsymbol{i} = 1$:

$$A\pi = b$$

$$A = \begin{bmatrix} \left(P^T - I\right) \\ \boldsymbol{i} \end{bmatrix}$$

$$A\pi = b$$

$$A = \begin{bmatrix} \left(P^T - I\right) \\ \boldsymbol{i} \end{bmatrix}$$

And **b** is a vector of which all elements except the last is 0.

Following from Rouché–Capelli,

$$\left(A^T A\right)\pi = \left(A^T\right) b$$

$$\left(A^T A\right)\pi = \left(A^T\right) b$$

from which as can solve for Pi, the stationary distribution, provided that the augmented matrix [A|b] rank is equal to the rank of the coefficient matrix A.

Again, this algorithm implementation can be made generic, extended, and implemented as a class.

Simulating from a Markov Chain

One can simulate from a Markov chain by noting that the collection of moves from any given state (the corresponding row in the probability matrix) form a multinomial distribution. One can thus simulate from a Markov Chain by simulating from a multinomial distribution.

One way to simulate from a multinomial distribution is to divide a line of length 1 into intervals proportional to the probabilities, and then picking an interval based on a uniform random number between 0 and 1.

See Wikipedia here https://en.wikipedia.org/wiki/Multinomial_distribution.

This is illustrated in the function simulate_multinomial below. We start with

$$R \sim U(0,1)$$

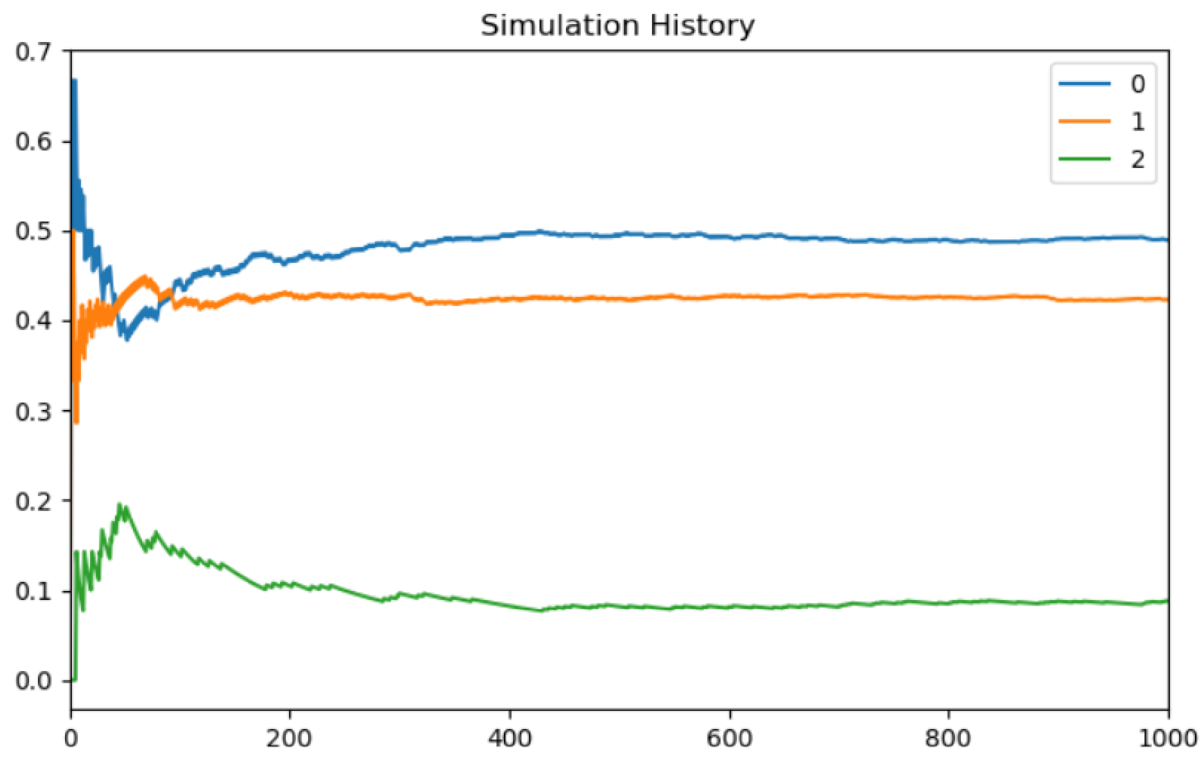
$$R \sim U(0,1)$$

We then use *cs*, the cumulative sum of probabilities in *P* in order to proportionally distribute random numbers.

```
import numpy as np
import pandas as pd
from random import seed
from random import random
import matplotlib.pyplot as pltP = np.array([[0.2, 0.7, 0.1],
              [0.9, 0.0, 0.1],
              [0.2, 0.8, 0.0]])stateChangeHist= np.array([[0.0,  0.0,
0.0],
                          [0.0, 0.0,  0.0],
                          [0.0, 0.0,  0.0]])state=np.array([[1.0, 0.0,
0.0]])
currentState=0
stateHist=state
dfStateHist=pd.DataFrame(state)
distr_hist = [[0,0,0]]
seed(4)# Simulate from multinomial distribution
def simulate_multinomial(vmultinomial):
  r=np.random.uniform(0.0, 1.0)
  CS=np.cumsum(vmultinomial)
  CS=np.insert(CS,0,0)
  m=(np.where(CS<r))[0]
  nextState=m[len(m)-1]
  return nextStatefor x in range(1000):
  currentRow=np.ma.masked_values((P[currentState]), 0.0)
  nextState=simulate_multinomial(currentRow)   # Keep track of state changes
stateChangeHist[currentState,nextState]+=1  # Keep track of the state
vector itself
```

```
  state=np.array([[0,0,0]])
  state[0,nextState]=1.0  # Keep track of state history
  stateHist=np.append(stateHist,state,axis=0)
  currentState=nextState  # calculate the actual distribution over the 3
states so far
  totals=np.sum(stateHist,axis=0)
  gt=np.sum(totals)
  distrib=totals/gt
  distrib=np.reshape(distrib,(1,3)
  distr_hist=np.append(distr_hist,distrib,axis=0)print(distrib)
P_hat=stateChangeHist/stateChangeHist.sum(axis=1)[:,None]
# Check estimated state transition probabilities based on history so
far:print(P_hat)dfDistrHist = pd.DataFrame(distr_hist)# Plot the
distribution as the simulation progresses over
timedfDistrHist.plot(title="Simulation History")
plt.show()
```

From the graph, it can be seen that the distribution starts converging to the stationary distribution somewhere after around 400 simulation steps.



The distribution converges to [0.47652348 0.41758242 0.10589411]:

The distribution is quite close to the stationary distribution that we calculated by solving the Markov chain earlier. In fact, rounded to two decimals it is identical: [0.49, 0.42, 0.09].

As we can see below, reconstructing the state transition matrix from the transition history gives us the expected result:

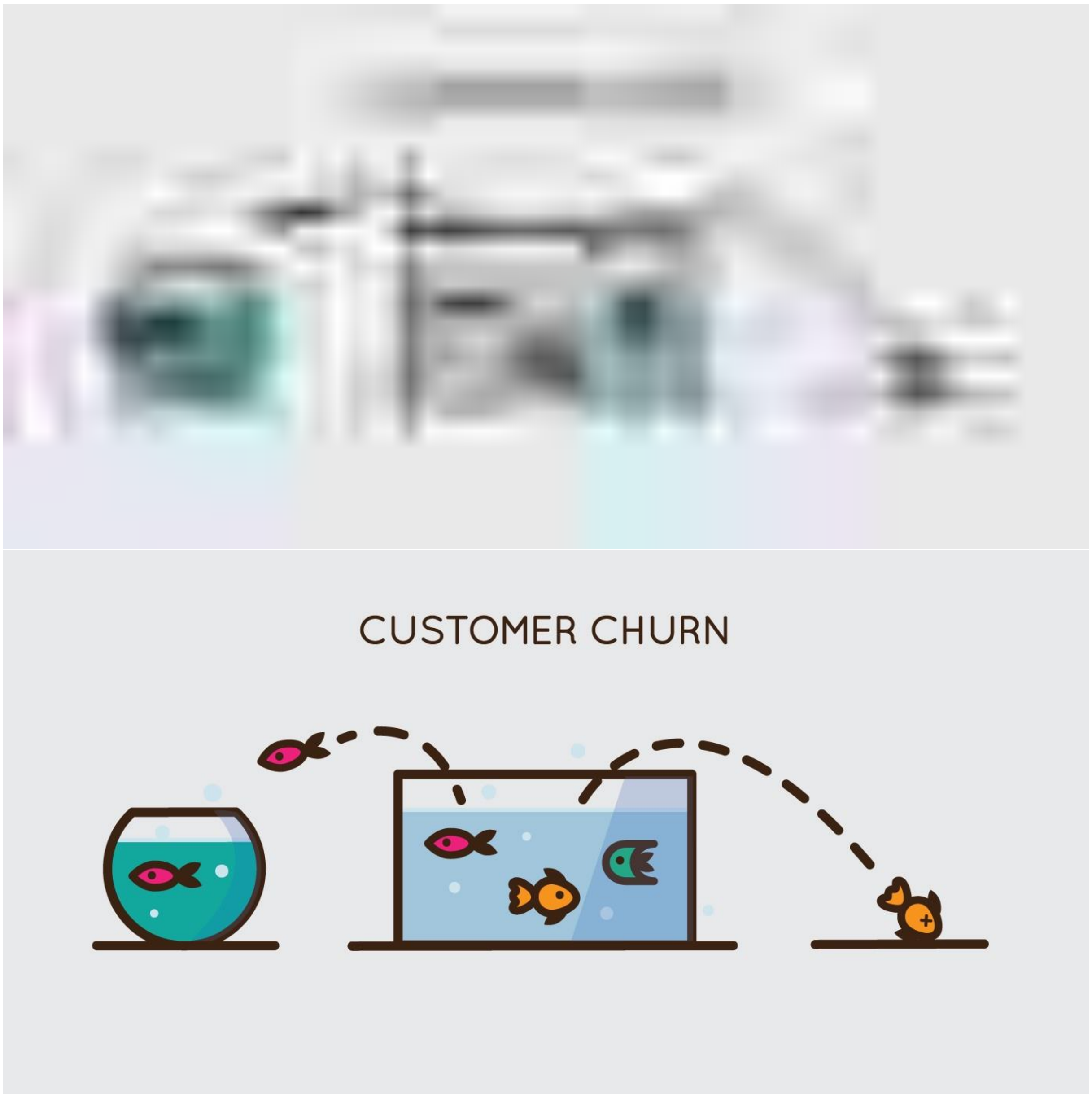
[0.18, 0.72, 0.10]
[0.91, 0.00, 0.09]
[0.19, 0.80, 0.00]

This algorithm implementation can be made generic, extended, and implemented as a class.

It illustrates how compact and concise algorithm implementation can be achieved with Python.

Application in Media, Telecommunications, or Similar Industry.

Let’s say, for a particular demographic, which aligns to high-value customers, we have 4 ‘competitors’ in a subscription media market (pay-TV, for example), with a relatively stable but changing distribution of [.55, 0.2, 0.1, 0.15], with the last group at 15% not having any particular subscription service, preferring to consume free content on demand.



https://www.livechatinc.com/blog/churn-rate/

The second biggest competitor (b) has just launched a new premium product and the incumbent suspects that it is eroding their market share. They would like to know how it may ultimately impact their market share if they do not intervene. They would also like to understand their own internal churn dynamics, and how it relates to their market share.

Let’s assume they know they sometimes lose customers to their competitors, including free content, particularly as their high Average Revenue Per User (ARPU) customer base evolves, and that they sometimes gain customers, but they do not understand the full picture.

So, we imagine they commission us to do a study.

Here we make a lot of implicit assumptions of the dynamics of the demographic in question to keep things simple. For example, we assume the transition probabilities remain constant.

Firstly, we do a market survey to understand how consumers are moving between the different providers, and from there we can construct a probability matrix as follows:

$$\begin{bmatrix} p_{aa} & p_{ab} & p_{ac} & p_{ad} \\ p_{ba} & p_{bb} & p_{bc} & p_{bd} \\ p_{ca} & p_{cb} & p_{cc} & p_{cd} \\ p_{da} & p_{db} & p_{dc} & p_{dd} \end{bmatrix}$$

$$\begin{bmatrix} p_{aa} & p_{ab} & p_{ac} & p_{ad} \\ p_{ba} & p_{bb} & p_{bc} & p_{bd} \\ p_{ca} & p_{cb} & p_{cc} & p_{cd} \\ p_{da} & p_{db} & p_{dc} & p_{dd} \end{bmatrix}$$

With a,b,c,d representing our market players.

The market research indicated produced the following estimated probabilities that a consumer would move from one service provider to another:

	A	B	C	D
A	0.9262	0.0385	0.01	0.0253
B	0.01	0.88	0.01	0.1
C	0.01	0.035	0.92	0.04
D	0.035	0.035	0.035	0.895

The first question we are interested in is what will happen if A keeps on losing customers to B at the estimated rate, taking into consideration all the other churn probabilities.

Using the matrix solution we derived earlier, and coding it in Python, we can calculate the new stationary distribution.

```
P = np.array([[0.9262, 0.0385, 0.01, 0.0253],
              [0.01, 0.94, 0.01, 0.04],
              [0.01, 0.035, 0.92, 0.04],
              [0.035, 0.035, 0.035, 0.895]])A=np.append(transpose(P)-
identity(4),[[1,1,1,1]],axis=0)b=transpose(np.array([0,0,0,0,1]))np.linalg.
solve(transpose(A).dot(A), transpose(A).dot(b))
```

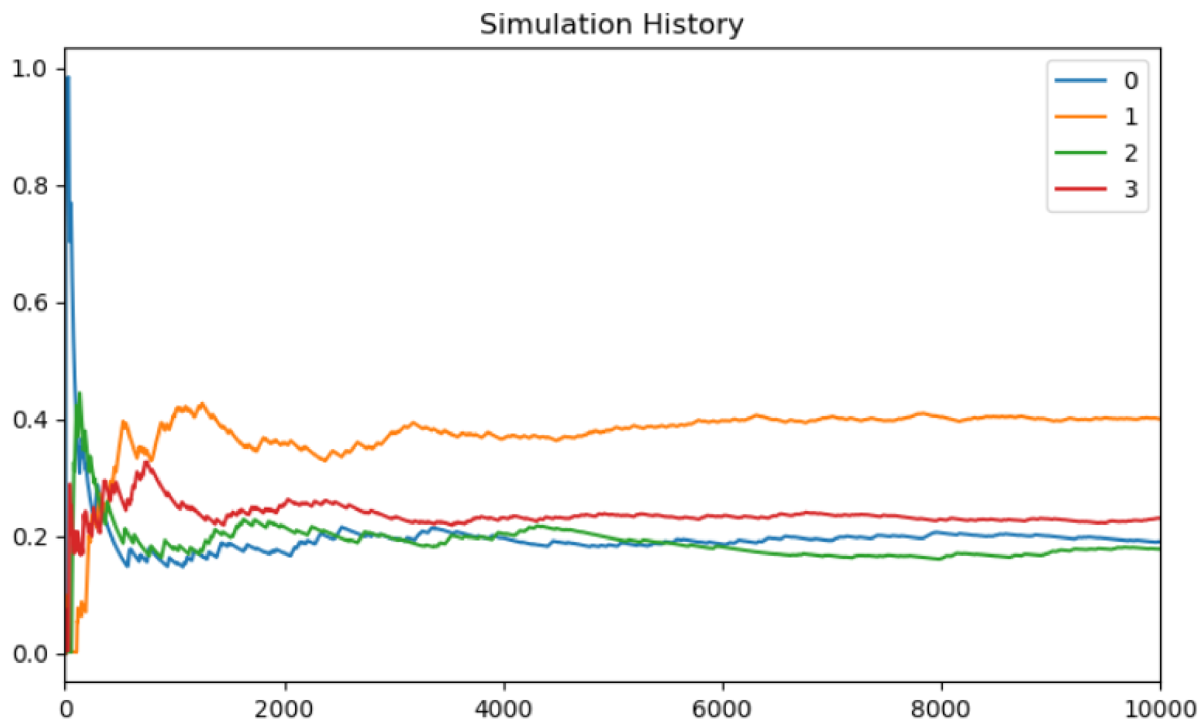
Which gives us the new stationary distribution [0.19, 0.37 , 0.18, 0.25]

However, when we check the rank of the coefficient matrix and the augmented matrix, we notice that, unlike the simpler example, they do not correspond. This means that the analytical problem formulation may not have a unique solution, so we want to check it with one of the other techniques.

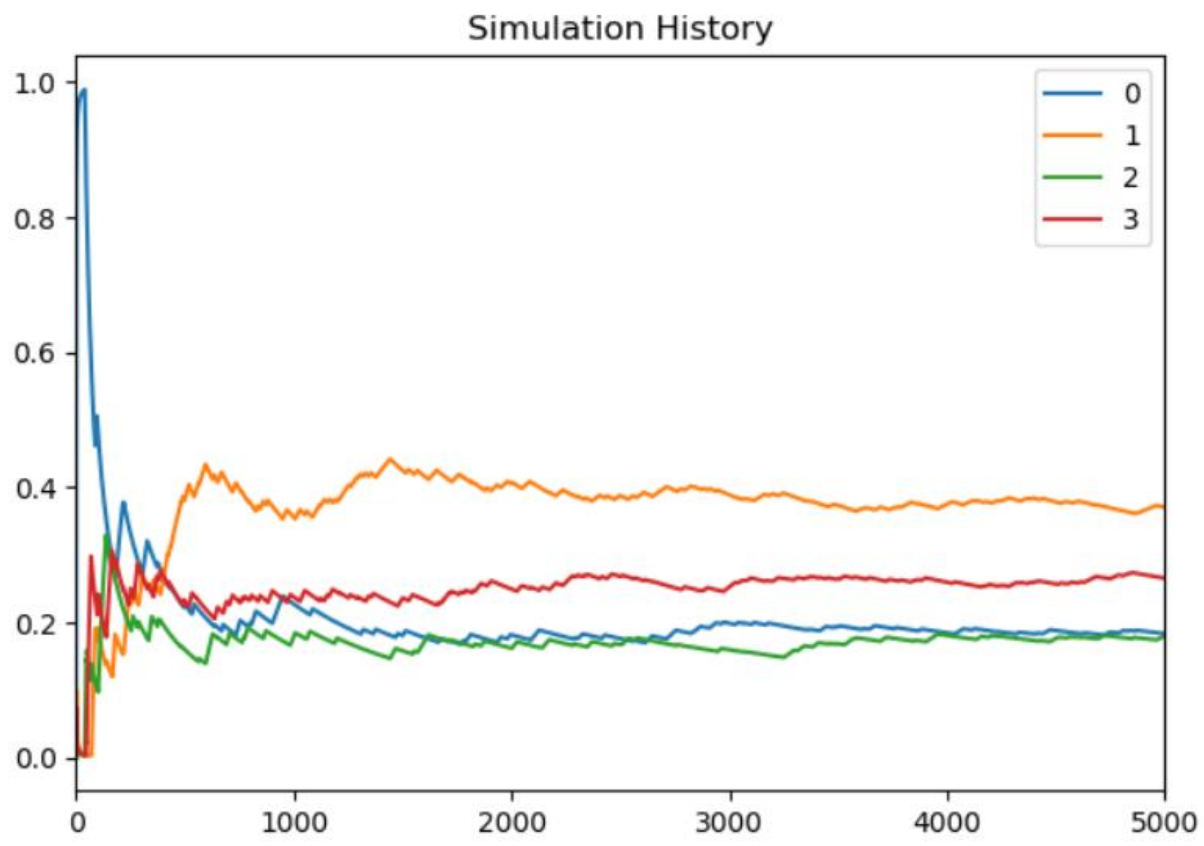
```
np.linalg.matrix_rank(np.append(A,np.transpose(b.reshape(1,5)), axis=1))
5

np.linalg.matrix_rank(A)
4
```

It can be shown that the iterative solution (where we raise the transition matrix to the power of n) does not converge, which leaves us with the simulation option.



Simulation History
0
1
2
3
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000



From the above, we can estimate that, in the long run, the stationary distribution will be something like this: [0.19, 0.4, 0.18, 0.23], which is actually very close to the analytical solution.

In other words, the market share of the incumbent can be expected to drop to around 20%, while the competitor will go up to around 40%.

From this, we can also see that the analytical and simulation solutions for the more complex problem, which is plausible in the real world, indeed still corresponds.

$$\begin{bmatrix} 0.910 & 0.040 & 0.01 & 0.035 \\ 0.010 & 0.940 & 0.01 & 0.034 \\ 0.010 & 0.035 & 0.094 & 0.029 \\ 0.030 & 0.034 & 0.042 & 0.910 \end{bmatrix}$$

$$\begin{bmatrix} 0.910 & 0.040 & 0.01 & 0.035 \\ 0.010 & 0.940 & 0.01 & 0.034 \\ 0.010 & 0.035 & 0.094 & 0.029 \\ 0.030 & 0.034 & 0.042 & 0.910 \end{bmatrix}$$

I hope you enjoyed this basic introduction on how discrete Markov Chains can be used to solve real-world problems and encourage you to think of questions in your own organization that can be answered in this way. You can also extend the example by calculating how valuable it would be to retain the churning customers, and thus how much it would be worth investing in retention.