Markov chains and their computation

Gavin Pownall

January 31, 2021

Abstract

The following decribes what a Markov chain is and how the state of a Markov Chain system is computed at any time step.

1 Markov Chain computation

A Markov chain is "a stochastic model desribing a sequence of events in which the probability of an event depends only on the state attained in the previous event" (see Wikipedia). This is explained with an example such as the one below.

Say that there are three possible states you can hold: A, B and C. If you are in A, the probability of moving from A to B is $M_{B,A}$, if you are in C the probability of moving to A is $M_{A,C}$. For any state i, the probability of staying in that same position for the next measurement is $M_{i,i}$. Etc. An example is given in Fig. 1.

The probabilities can be represented by a *stochastic* matrix $M_{i,j}$, where (A, B, C) maps to $(1, 2, 3)$. For Fig. 1, we have the following matrix:

$$\begin{pmatrix} 1/3 & 1/2 & 1/10 \\ 1/3 & 1/4 & 2/10 \\ 1/3 & 1/4 & 7/10 \end{pmatrix} \tag{1}$$

Notice how, for each column, the elements total one.

We also need to define an initial state I, which will simply be the probability of being in each state at time step 0, for example $(1/3, 1/3, 1/3)$ or $(1, 0, 0)$. Again, this must sum to one. We label this P_0.

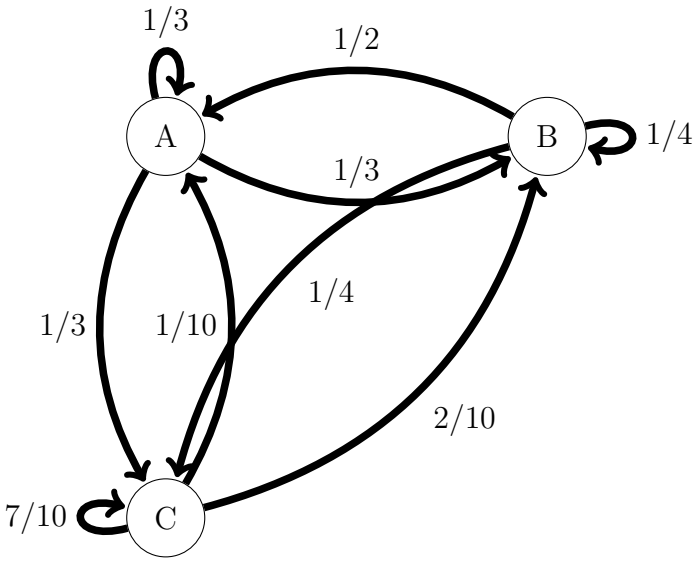

Figure 1: A display of a 3-state Markov-chain process.

The probability of being in any step at time t is

$$P_t = MP_{t-1} \tag{2}$$

In other words, by the definition that this is a Markov-chain process, the probability matrix is multiplied by the previous state in order to give the current state. Given P_0 and M, we can therefore calculate the probability of the state at any time t in one line:

$$P_t = M^t P_0 \tag{3}$$

Obviously this isn't ideal to do by hand, and even for a computer this could get expensive for a large number of states or large values of t. For this reason, we diagonalise the M matrix – or, in simpler terms, express P_0 in terms of the eigenvectors of M.

Define the eigenvalues of M to be m_i and the eigenvectors to be e_i. The initial state is written in terms of the eigenvectors to give:

$$
\begin{aligned}
P_t &= M^t P_0 \\
&= \sum_{n=1}^{i} M^t c_n e_n \\
&= \sum_{n=1}^{i} c_n m_n^t e_n,
\end{aligned}
\tag{4}
$$

where c_i are scalar values, $P_0 = \sum_{n=1}^{i} c_n e_n$. This is far faster to compute.

Notes:

- It is given without proof that the eigenvectors of a stochastic matrix span the full space they inhabit.
- It is given, again without proof, that one of the eigenvalues is 1 and that the others are less than 1. As a result, clearly from the above equation, as $t \to \infty$, all of the terms in the sum vanish except for one. This is known as the *stationary state*.

2 Python code

The class for computing all of the above is given in MarkovClass.py. By running this script directly, the above example is computed.

```
python3 MarkovClass.py

Initial state:  [1 0 0]
Probability matrix:
  [[0.33333333 0.5        0.1       ]
 [0.33333333 0.25       0.2       ]
 [0.33333333 0.25       0.7       ]]
Constants:  [ 0.49382716 -0.42910033  0.43247949]
Eigenvalues:  [ 1.          0.39013419 -0.10680086]
Eigenvectors:  [[ 0.525       0.5         1.        ]
 [-0.7183897  -0.2816103   1.        ]
 [ 1.         -0.85033548 -0.14966452]]
Equation with time:
P(t) = 0.4938271604938267*(1.0000000000000009^t)*[0.525 0.5   1.   ]
```

```
+-0.42910032543113297*(0.39013419193125953^t)*[-0.7183897 -0.2816103  1.       ]
+0.43247948804680914*(-0.10680085859792622^t)*[ 1.         -0.85033548 -0.14966452
Stationary state: [0.25925926 0.24691358 0.49382716]
```

There are clearly issues with floating point errors, but these come directly from the numpy linear algebra computation and unfortunately cannot be helped.

3 Example usages

3.1 Gambler's ruin

The Gambler's ruin problem is well known. Given a starting value, x, a value at which the gambler would "cash out", y, and a probability of increasing by 1 and decreasing by 1, what is the probability that the gambler would eventually cash out?

This can be solved using an iterative derivation. For example, for a fair coin the probability of ending penniless is:

$$P = \frac{x/y}{.} \tag{5}$$

Using Markov chains, not only can this be calculated computationally, but you can also calculate the probability of the gambler having a certain number of coins at any time step. See GamblersRuin.py for more details.

3.2 Monopoly

The monopoly game can also be simulated using Markov chain analysis. In the Monopoly.py script, a simple setup is created with dice rolls and chance/community chest "advance to" cards accounted for. The stationary state gives the optimal position for properties. As has been observed elsewhere, the jail square is the most likely, followed by those around 7 squared afterward.