

Predicting future states using Markov Chains

Pietro Mascolo

Optum Ireland Ltd.

April 21, 2018




What I'm going to tell you

- Introduction;
- definition and representation of Markov Chains;
- how Markov Chains can be used;
- a bit of maths;
- a bit of code;
- a small demo...





Who I am


- **Mad scientist**;
- **Python enthusiast**;
- **Kotlin**/Scala practitioner;
- **Golang** padawan;
- Ham Radio Operator (EI/IZ4VVE);
- Mountain hiker;
- Karateka;
- Amateur Photographer;
- ...





Pietro Mascolo
Data Scientist

 Ireland

 pietro@mascolo.eu


 [pmascolo](#)

 iz4vve


 [@iz4vve](#)


Who I am


- **Mad-scientist** Physicist;
- **Python** enthusiast;
- **Kotlin**/Scala practitioner;
- **Golang** padawan;
- Ham Radio Operator (EI/IZ4VVE);
- Mountain hiker;
- Karateka;
- Amateur Photographer;
- ...




Pietro Mascolo
Data Scientist

 Ireland

 pietro@mascolo.eu

 [pmascolo](#)

 iz4vve

 [@iz4vve](#)

UNITEDHEALTH GROUP

Ranked 6th of the Fortune 500

~\$200B FY17E revenue



Health Benefits

A diversified enterprise with
complementary but distinct
business platforms



Health Services

OUR MISSION

Helping people live healthier lives and helping make the health system work better for everyone

OUR VALUES

Integrity

Compassion

Relationships

Innovation

Performance

© 2017 Optum, Inc. All rights reserved. As of Q3, 2017.



Why all this?

Problem statement (sorry for having to "anonymise" everything...)

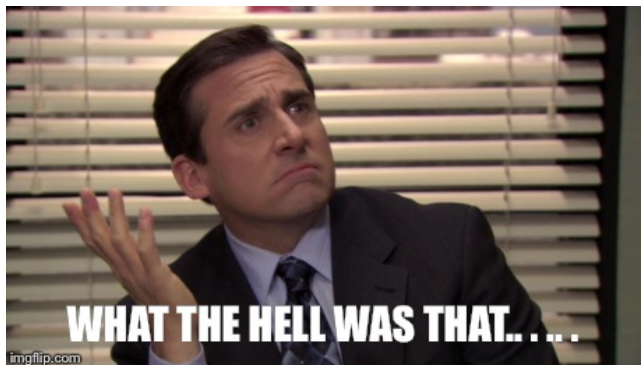
For each user of a system, a sequence of events occurs. Some sequences are good, some are bad. We want to determine driving factors of bad sequences as well as when/where they occur in the system, and for which users.

Definition

A Markov chain is collection of random variables X_t having the property that, given the present, the future is conditionally independent of the past.

$$P(X_t = j | X_0 = i_0, X_1 = i_1, \dots, X_{t-1} = i_{t-1}) = P(X_t = j | X_{t-1} = i_{t-1})$$

Markov Chains - definition and representation 1/2



Definition

A Markov chain is collection of random variables X_t having the property that, given the present, the future is conditionally independent of the past.

$$P(X_t = j | X_0 = i_0, X_1 = i_1, \dots, X_{t-1} = i_{t-1}) = P(X_t = j | X_{t-1} = i_{t-1})$$

Don't panic!

Everything will be clearer with an example...

Definition

A Markov chain is collection of random variables X_t having the property that, given the present, the future is conditionally independent of the past.

$$P(X_t = j | X_0 = i_0, X_1 = i_1, \dots, X_{t-1} = i_{t-1}) = P(X_t = j | X_{t-1} = i_{t-1})$$

Let us imagine we have the following sequence:

1, 2, 1, 2, 1, 2, 3, 1, 2

Definition

A Markov chain is collection of random variables X_t having the property that, given the present, the future is conditionally independent of the past.

$$P(X_t = j | X_0 = i_0, X_1 = i_1, \dots, X_{t-1} = i_{t-1}) = P(X_t = j | X_{t-1} = i_{t-1})$$

Let us imagine we have the following sequence:

1, 2, 1, 2, 1, 2, 3, 1, 2

Based on these numbers we can say that:

- If the current state is 1, there is a 100% probability of moving to state 2;
- if the current state is 2, there is 66% probability of evolving to state 1 and 33% of evolving to state 3;
- if the current state is 3, there is a 100% probability of evolving to state 1.

Markov Chains - definition and representation 2/2

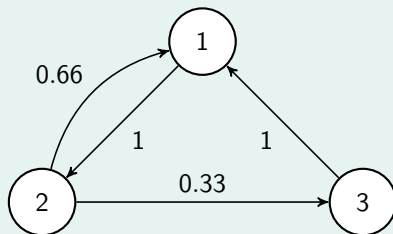
The states we have determined can be represented in a more visual fashion by using a matrix (or - even better - a graph).

Two representations

Matrix representation

$$T = \begin{pmatrix} 0 & 1 & 0 \\ 0.66 & 0 & 0.33 \\ 1 & 0 & 0 \end{pmatrix}$$

Graph representation



A bit of Maths 1/3 - description

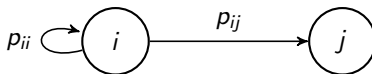
Now we need a bit of maths to show what we can do with Markov Chains...



A bit of Maths 1/3 - description

We can describe a Markov Chain as follows:

Given a set of possible states: $S = \{s_1, s_2, \dots, s_n\}$, at each step, a generic state i can move to a new state j or remain in the same initial state with certain *transition probabilities*:



A bit of Maths 2/3 - transition probabilities and next state

All the transition probabilities can be collected in a **transition matrix**:

$$T = \begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \\ p_{n1} & & p_{nn} \end{pmatrix}$$

A bit of Maths 2/3 - transition probabilities and next state

All the transition probabilities can be collected in a **transition matrix**:

$$T = \begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \\ p_{n1} & & p_{nn} \end{pmatrix}$$

And given an initial state, the next state transition probabilities are given by:

$$\mathbf{S}_{i+1} = \mathbf{S}_i^T \times \mathbf{T}$$

and both S_i and S_{i+1} can be either a definite state (vector of all zeros and a single 1) or a probabilistic state (vector of decimals summing up to 1).

A bit of Maths 3/3 - High order Markov Chains

So far we only treated first order Markov Chains: now we can expand on that!
What will the next state be, given that the current state is j **AND** the previous state was i ?

The transition matrix can be computed the same way as before, only considering all subsequent pairs in the sequence of states.

A bit of Maths 3/3 - High order Markov Chains

In case of a second order chain, the transition matrix will be of size $N(N - 1) \times N(N - 1)$, where N is the cardinality of the set of possible states, and it will relate pairs of states instead of single states.

$$T = \begin{pmatrix} p_{(11)(11)} & \cdots & p_{(11)(1n)} \\ \vdots & \ddots & \\ p_{(n1)(11)} & & p_{(nn)(nn)} \end{pmatrix}$$

WARNING!

For clarity, I will omit a lot of stuff from the code (error handling, correct matrix handling, logging, dosctrings, ...).

You should NOT run the code as is: IT WON'T WORK!

Markov Chain object implementation

Let's start simple, shall we?



Markov Chain object implementation

Let's start simple, shall we?

```
class MarkovChain(object):  
    pass
```

Markov Chain object implementation

Let's start simple, shall we?

```
class MarkovChain(object):  
    pass
```



Initialisation

We need to initialise the number of states and the transition matrix:

Initialisation

We need to initialise the number of states and the transition matrix:

```
def __init__(self, n_states, order=1):  
    self.number_of_states = n_states  
    self.order = order
```


Initialisation

We need to initialise the number of states and the transition matrix:

```
def __init__(self, n_states, order=1):
    self.number_of_states = n_states
    self.order = order

    self.possible_states = {
        j: i for i, j in
            enumerate(
                itertools.product(range(n_states),
                                repeat=order)
            )
    }
```

Initialisation

We need to initialise the number of states and the transition matrix:

```
def __init__(self, n_states, order=1):
    self.number_of_states = n_states
    self.order = order

    self.possible_states = {
        j: i for i, j in
            enumerate(
                itertools.product(range(n_states),
                                repeat=order)
            )
    }

    self.transition_matrix = sparse.dok_matrix((
        len(self.possible_states), len(self.possible_states))
    ), dtype=np.float64)
```

Transition update

Having initialised our matrix T , we need to update it when we examine a sequence:

Transition update

Having initialised our matrix T , we need to update it when we examine a sequence:

```
def update_transition_matrix(self, states_sequence):  
  
    visited_states = [  
        states_sequence[i: i + self.order]  
        for i in range(len(states_sequence) - self.order + 1)  
    ]
```

Transition update

Having initialised our matrix T , we need to update it when we examine a sequence:

```
def update_transition_matrix(self, states_sequence):  
  
    visited_states = [  
        states_sequence[i: i + self.order]  
        for i in range(len(states_sequence) - self.order + 1)  
    ]  
  
    for state_index, i in enumerate(visited_states):  
        self.transition_matrix[  
            self.possible_states[tuple(i)],  
            self.possible_states[tuple(visited_states[  
                state_index + self.order  
            ])]  
        ] += 1
```

Something is missing though...

Something is missing though...

Something is still missing though...

This matrix represents probabilities: **ROWS MUST SUM TO 1!**

This matrix represents probabilities: **ROWS MUST SUM TO 1!**

```
def normalise_transitions(self):  
    self.transition_matrix = preprocessing.normalize(  
        self.transition_matrix, norm="l1"  
    )
```

Calling `update_transition_matrix` on a sequence of sequences:

Calling `update_transition_matrix` on a sequence of sequences:

```
def fit(self, state_sequences):  
    for index, sequence in enumerate(state_sequences):  
        self.update_transition_matrix(sequence)  
    self.normalize_transitions()
```

Predicting next state

Now we need the predict method...



Predicting next state

Now we need the predict method... How do we implement that?



Predicting next state

Remember:

$$S_{i+1} = S_i^T \times T$$

Predicting next state

Remember:

$$S_{i+1} = S_i^T \times T$$

But what about a generic number of time steps?

$$S_{i+N} = ???$$

Predicting next state

Remember:

$$S_{i+1} = S_i^T \times T$$

But what about a generic number of time steps?

$$S_{i+N} = S_{i+N-1}^T \times T$$

Remember:

$$S_{i+1} = S_i^T \times T$$

But what about a generic number of time steps?

$$S_{i+N} = (S_{i+N-2}^T \times T)^T \times T$$

Remember:

$$S_{i+1} = S_i^T \times T$$

But what about a generic number of time steps?

$$S_{i+N} = (S_{i+N-2}^T \times T)^T \times T$$

We have to take this all the way down to S_i .
It looks a bit unwieldy to implement...

Oh, wait!

If we **DO** take it all the way to S_i we're only left with T multiplied by itself N times!

Predicting next state

Remember:

$$S_{i+1} = S_i^T \times T$$

But what about a generic number of time steps?

$$S_{i+N} = (S_{i+N-2}^T \times T)^T \times T$$

We have to take this all the way down to S_i .
It looks a bit unwieldy to implement...

Oh, wait!

If we **DO** take it all the way to S_i we're only left with T multiplied by itself N times!

$$S_{i+N} = S_i^T \times T^N$$

Now, **this** we can implement :D

Predicting next state

```
def predict_state(self, current_state, num_steps=1):  
    _next_state = sparse.csr_matrix(current_state).dot(  
        np.power(self.transition_matrix, num_steps)  
    )  
  
    return _next_state
```

There...

There we go!

And that's it for the (simplified) implementation... We're good to go!



Demo!



You've made it through!!

Thanks for your attention!

pietro_mascolo@optum.com

@iz4vve

Slides and code: <http://bit.ly/2H8giAW>

