# mchmm

Maksim Terpilowski

Jun 02, 2021

# USAGE

*mchmm* is a Python package implementing Markov chains and Hidden Markov models in pure NumPy and SciPy. It can also visualize Markov chains.

CHAPTER

ONE

# INSTALLATION

## 1.1 PyPi

```
pip install mchmm
```

## 1.2 GitHub

```
git clone https://github.com/maximtrp/mchmm.git
cd mchmm
pip install . --user
```

CHAPTER

# TWO

# TUTORIALS

## 2.1 Discrete Markov chains

Initializing a Markov chain using some data.

```
>>> import mchmm as mc
>>> a = mc.MarkovChain().from_data(
↪'AABCABCBAAAACBCBACBABCABCBACBACBABABCBACBBCBBCBCBCBACBABABCBCBAAACABABCBBCBCBCBCBAABCBBCBCBCCCBABCB
↪')
```

Now, we can look at the observed transition *frequency* matrix:

```
>>> a.observed_matrix
array([[ 7., 18.,  7.],
       [19.,  5., 29.],
       [ 5., 30.,  3.]])
```

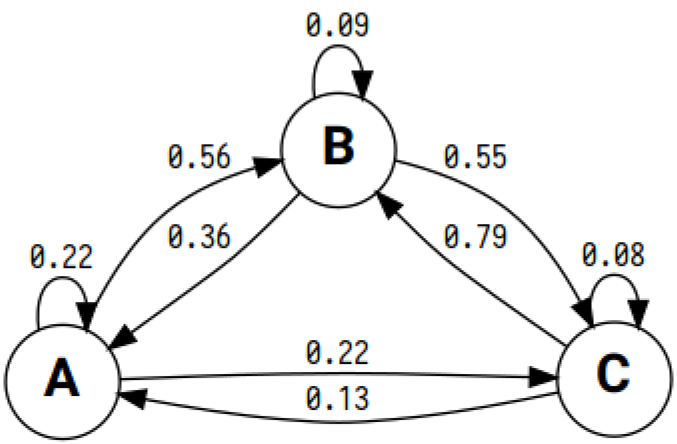
And the observed transition *probability* matrix:

```
>>> a.observed_p_matrix
array([[0.21875   , 0.5625    , 0.21875   ],
       [0.35849057, 0.09433962, 0.54716981],
       [0.13157895, 0.78947368, 0.07894737]])
```

You can visualize your Markov chain. First, build a directed graph with `graph_make()` method of `MarkovChain` object. Then `render()` it.

```
>>> graph = a.graph_make(
      format="png",
      graph_attr=[("rankdir", "LR")],
      node_attr=[("fontname", "Roboto bold"), ("fontsize", "20")],
      edge_attr=[("fontname", "Iosevka"), ("fontsize", "12")]
    )
>>> graph.render()
```

Here is the result:




Pandas can help us annotate columns and rows:

```
>>> import pandas as pd
>>> pd.DataFrame(a.observed_matrix, index=a.states, columns=a.states, dtype=int)
    A   B   C
A   7  18   7
B  19   5  29
C   5  30   3
```

Viewing the expected transition frequency matrix:

```
>>> a.expected_matrix
array([[ 8.06504065, 13.78861789, 10.14634146],
       [13.35772358, 22.83739837, 16.80487805],
       [ 9.57723577, 16.37398374, 12.04878049]])
```

Calculating Nth order transition probability matrix:

```
>>> a.n_order_matrix(a.observed_p_matrix, order=2)
array([[0.2782854 , 0.34881028, 0.37290432],
       [0.1842357 , 0.64252707, 0.17323722],
       [0.32218957, 0.21081868, 0.46699175]])
```

Carrying out a chi-squared test:

```
>>> a.chisquare(a.observed_matrix, a.expected_matrix, axis=None)
Power_divergenceResult(statistic=47.89038802624337, pvalue=1.0367838347591701e-07)
```

Finally, let's simulate a Markov chain given our data.

```
>>> ids, states = a.simulate(10, start='A', seed=np.random.randint(0, 10, 10))
>>> ids
array([0, 2, 1, 0, 2, 1, 0, 2, 1, 0])
>>> states
array(['A', 'C', 'B', 'A', 'C', 'B', 'A', 'C', 'B', 'A'], dtype='<U1')
>>> "".join(states)
'ACBACBACBA'
```

## 2.2 Hidden Markov models

We will use a fragment of DNA sequence with TATA box as an example. Initializing a hidden Markov model with sequences of observations and states:

```
>>> import mchmm as mc
>>> obs_seq = 'AGACTGCATATATAAGGGGCAGGCTG'
>>> sts_seq = '00000000111111100000000000'
>>> a = mc.HiddenMarkovModel().from_seq(obs_seq, sts_seq)
```

Unique states and observations are automatically inferred:

```
>>> a.states
['0' '1']
>>> a.observations
['A' 'C' 'G' 'T']
```

The transition probability matrix for all states can be accessed using `tp` attribute:

```
>>> a.tp
[[0.94444444 0.05555556]
 [0.14285714 0.85714286]]
```

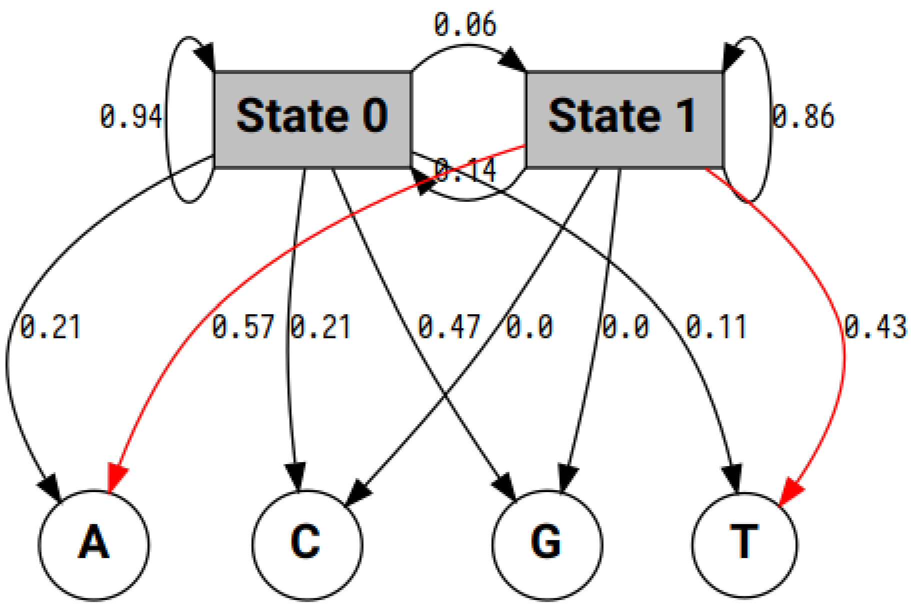
There is also `ep` attribute for the emission probability matrix for all states and observations.

```
>>> a.ep
[[0.21052632 0.21052632 0.47368421 0.10526316]
 [0.57142857 0.         0.         0.42857143]]
```

Converting the emission matrix to Pandas DataFrame:

```
>>> import pandas as pd
>>> pd.DataFrame(a.ep, index=a.states, columns=a.observations)
          A         C         G         T
0  0.210526  0.210526  0.473684  0.105263
1  0.571429  0.000000  0.000000  0.428571
```

Directed graph of the hidden Markov model:




Graph can be visualized using `graph_make` method of `HiddenMarkovModel` object:

```
>>> graph = a.graph_make(
      format="png",
      graph_attr=[("rankdir", "LR"), ("ranksep", "1"), ("rank", "same")]
    )
>>> graph.render()
```

### 2.2.1 Viterbi algorithm

Running Viterbi algorithm on new observations.

```
>>> new_obs = "GGCATTGGGCTATAAGAGGAGCTTG"
>>> vs, vsi = a.viterbi(new_obs)
>>> # states sequence
>>> print("VI", "".join(vs))
>>> # observations
>>> print("NO", new_obs)
```

```
VI 0000000001111100000000000
NO GGCATTGGGCTATAAGAGGAGCTTG
```

### 2.2.2 Baum-Welch algorithm

Using Baum-Welch algorithm to infer the parameters of a Hidden Markov model:

```
>>> obs_seq = 'AGACTGCATATATAAGGGGCAGGCTG'
>>> a = hmm.HiddenMarkovModel().from_baum_welch(obs_seq, states=['0', '1'])
>>> # training log: KL divergence values for all iterations
>>> a.log
```

```
{
  'tp': [0.008646969455670256, 0.0012397829805491124, 0.0003950986109761759],
  'ep': [0.09078874423746826, 0.0022734816599056084, 0.0010118204023946836],
  'pi': [0.009030829793043593, 0.016658391248503462, 0.0038894983546756065]
}
```

The inferred transition (*tp*), emission (*ep*) probability matrices and initial state distribution (*pi*) can be accessed as shown:

```
>>> a.ep, a.tp, a.pi
```

This model can be decoded using Viterbi algorithm:

```
>>> new_obs = "GGCATTGGGCTATAAGAGGAGCTTG"
>>> vs, vsi = a.viterbi(new_obs)
>>> print("VI", "".join(vs))
>>> print("NO", new_obs)
```

```
VI 0011100001111100000001100
NO GGCATTGGGCTATAAGAGGAGCTTG
```

# CHAPTER THREE

# MCHMM.MARKOVCHAIN

**class** `mchmm.MarkovChain`(*states: Optional[Union[list, numpy.ndarray]] = None*, *obs: Optional[Union[list, numpy.ndarray]] = None*, *obs_p: Optional[Union[list, numpy.ndarray]] = None*)

Bases: `object`

**`__init__`**(*states: Optional[Union[list, numpy.ndarray]] = None*, *obs: Optional[Union[list, numpy.ndarray]] = None*, *obs_p: Optional[Union[list, numpy.ndarray]] = None*)

Discrete Markov Chain.

**Parameters**

- **states** (*Optional[Union[numpy.ndarray, list]]*) – State names list.
- **obs** (*Optional[Union[numpy.ndarray, list]]*) – Observed transition frequency matrix.
- **obs_p** (*Optional[Union[numpy.ndarray, list]]*) – Observed transition probability matrix.

**`_transition_matrix`**(*seq: Optional[Union[str, numpy.ndarray, list]] = None*, *states: Optional[Union[str, numpy.ndarray, list]] = None*) → numpy.ndarray

Calculate a transition frequency matrix.

**Parameters**

- **seq** (*Optional[Union[str, list, numpy.ndarray]]*) – Observations sequence.
- **states** (*Optional[Union[str, list, numpy.ndarray]]*) – List of states.

**Returns** **matrix** – Transition frequency matrix.

**Return type** numpy.ndarray

**`chisquare`**(*obs: Optional[numpy.ndarray] = None*, *exp: Optional[numpy.ndarray] = None*, ***kwargs*) → Tuple[Union[float, numpy.ndarray], Union[float, numpy.ndarray]]

Wrapper function for carrying out a chi-squared test using *scipy.stats.chisquare* method.

**Parameters**

- **obs** (*numpy.ndarray*) – Observed transition frequency matrix.
- **exp** (*numpy.ndarray*) – Expected transition frequency matrix.
- **kwargs** (*optional*) – Keyword arguments passed to *scipy.stats.chisquare* method.

**Returns**

- **chisq** (*float or numpy.ndarray*) – Chi-squared test statistic.
- **p** (*float or numpy.ndarray*) – P value of the test.

`from_data`(*seq: Union[str, numpy.ndarray, list]*) → object
Infer a Markov chain from data. States, frequency and probability matrices are automatically calculated and assigned to as class attributes.

**Parameters** `seq` (*Union[str, np.ndarray, list]*) – Sequence of events. A string or an array-like object exposing the array interface and containing strings or ints.

**Returns** **MarkovChain** – Trained MarkovChain class instance.

**Return type** object

`graph_make`(**args*, ***kwargs*) → graphviz.dot.Digraph
Make a directed graph of a Markov chain using *graphviz*.

**Parameters**

- `args` (*optional*) – Arguments passed to the underlying *graphviz.Digraph* method.
- `kwargs` (*optional*) – Keyword arguments passed to the underlying *graphviz.Digraph* method.

**Returns** **graph** – Digraph object with its own methods.

**Return type** graphviz.dot.Digraph

**Note:** *graphviz.dot.Digraph.render* method should be used to output a file.

`n_order_matrix`(*mat: Optional[numpy.ndarray] = None*, *order: int = 2*) → numpy.ndarray
Create Nth order transition probability matrix.

**Parameters**

- `mat` (*numpy.ndarray, optional*) – Observed transition probability matrix.
- `order` (*int, optional*) – Order of transition probability matrix to return. Default is 2.

**Returns** **x** – Nth order transition probability matrix.

**Return type** numpy.ndarray

`prob_to_freq_matrix`(*mat: Optional[numpy.ndarray] = None*, *row_totals: Optional[numpy.ndarray] = None*) → numpy.ndarray
Calculate a transition frequency matrix given a transition probability matrix and row totals. This method is meant to be used to calculate a frequency matrix for a Nth order transition probability matrix.

**Parameters**

- `mat` (*numpy.ndarray, optional*) – Transition probability matrix.
- `row_totals` (*numpy.ndarray, optional*) – Row totals of transition frequency matrix.

**Returns** **x** – Transition frequency matrix.

**Return type** numpy.ndarray

`simulate`(*n: int*, *tf: Optional[numpy.ndarray] = None*, *states: Optional[Union[list, numpy.ndarray]] = None*, *start: Optional[Union[str, int]] = None*, *ret: str = 'both'*, *seed: Optional[Union[list, numpy.ndarray]] = None*) → Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]
Markov chain simulation based on *scipy.stats.multinomial*.

**Parameters**

- `n` (*int*) – Number of states to simulate.

- **tf** (*numpy.ndarray, optional*) – Transition frequency matrix. If None, *observed_matrix* instance attribute is used.
- **states** (*Optional[Union[np.ndarray, list]]*) – State names. If None, *states* instance attribute is used.
- **start** (*Optional[str, int]*) – Event to begin with. If integer is passed, the state is chosen by index. If string is passed, the state is chosen by name. If *random* string is passed, a random state is taken. If left unspecified (None), an event with maximum probability is chosen.
- **ret** (*str, optional*) – Return state indices if *indices* is passed. If *states* is passed, return state names. Return both if *both* is passed.
- **seed** (*Optional[Union[list, numpy.ndarray]]*) – Random states used to draw random variates (of size *n*). Passed to *scipy.stats.multinomial* method.

**Returns**

- **x** (*numpy.ndarray*) – Sequence of state indices.
- **y** (*numpy.ndarray, optional*) – Sequence of state names. Returned if *return* arg is set to 'states' or 'both'.

CHAPTER

# FOUR

# MCHMM.HIDDENMARKOVMODEL

class mchmm.**HiddenMarkovModel**(*observations: Optional[Union[list, numpy.ndarray]] = None*, *states: Optional[Union[list, numpy.ndarray]] = None*, *tp: Optional[Union[list, numpy.ndarray]] = None*, *ep: Optional[Union[list, numpy.ndarray]] = None*, *pi: Optional[Union[list, numpy.ndarray]] = None*)

Bases: `object`

**__init__**(*observations: Optional[Union[list, numpy.ndarray]] = None*, *states: Optional[Union[list, numpy.ndarray]] = None*, *tp: Optional[Union[list, numpy.ndarray]] = None*, *ep: Optional[Union[list, numpy.ndarray]] = None*, *pi: Optional[Union[list, numpy.ndarray]] = None*)

Hidden Markov model.

**Parameters**

- **observations** (*Optional[Union[list, np.ndarray]]*) – Observations space (of size N).
- **states** (*Optional[Union[list, np.ndarray]]*) – List of states (of size K).
- **tp** (*Optional[Union[list, np.ndarray]]*) – Transition matrix of size K × K which stores transition probability of transiting from state i (row) to state j (col).
- **ep** (*Optional[Union[list, np.ndarray]]*) – Emission matrix of size K × N which stores probability of seeing observation j (col) from state i (row). N is the length of observation space O = [o_1, o_2, …, o_N].
- **pi** (*Optional[Union[list, np.ndarray]]*) – Initial state probabilities array (of size K).

**_emission_matrix**(*obs_seq: Optional[Union[str, numpy.ndarray, list]] = None*, *states_seq: Optional[Union[str, numpy.ndarray, list]] = None*, *obs: Optional[Union[str, numpy.ndarray, list]] = None*, *states: Optional[Union[str, numpy.ndarray, list]] = None*) → numpy.ndarray

Calculate an emission probability matrix.

**Parameters**

- **obs_seq** (*str or array_like*) – Sequence of observations (of size N). Observation space = [o_1, o_2, …, o_N].
- **states_seq** (*str or array_like*) – Sequence of states (of size K). State space = [s_1, s_2, …, s_K].

**Returns** **ep** – Emission probability matrix of size K × N.

**Return type** numpy.ndarray

**_transition_matrix**(*seq: Optional[Union[str, numpy.ndarray, list]] = None*, *states: Optional[Union[str, numpy.ndarray, list]] = None*)

Calculate a transition probability matrix which stores transition probability of transiting from state i to state j.

**Parameters**

- **seq** (*Optional[Union[str, numpy.ndarray, list]]*) – Sequence of states.
- **states** (*Optional[Union[str, numpy.ndarray, list]]*) – List of unique states.

**Returns** **matrix** – Transition frequency matrix.

**Return type** numpy.ndarray

**from_baum_welch**(*obs_seq: Union[str, list, numpy.ndarray]*, *states: Optional[Union[list, numpy.ndarray]] = None*, *thres: Optional[float] = 0.001*, *obs: Optional[Union[str, numpy.ndarray, list]] = None*, *tp: Optional[numpy.ndarray] = None*, *ep: Optional[numpy.ndarray] = None*, *pi: Optional[Union[list, numpy.ndarray]] = None*) → object

Baum-Welch algorithm.

**Parameters**

- **obs_seq** (*Union[str, list, numpy.ndarray]*) – Sequence of observations.
- **states** (*Optional[Union[list, numpy.ndarray]]*) – List of states (of size K).
- **thres** (*Optional[float]*) – Convergence threshold. Kullback-Leibler divergence value below which model training is stopped.
- **obs** (*Optional[Union[list, numpy.ndarray]]*) – Observations space (of size N).
- **tp** (*Optional[numpy.ndarray]*) – Transition matrix (of size K × K) which stores transition probability of transiting from state i (row) to state j (col).
- **ep** (*Optional[numpy.ndarray]*) – Emission matrix (of size K × N) which stores probability of seeing observation j (col) from state i (row). N is the length of observation space, O = {o_1, o_2, …, o_N}.
- **pi** (*Optional[Union[list, numpy.ndarray]]*) – Initial probabilities array (of size K).

**Returns** Hidden Markov model trained using Baum-Welch algorithm.

**Return type** *HiddenMarkovModel*

**from_seq**(*obs_seq: Union[str, list, numpy.ndarray]*, *states_seq: Union[str, list, numpy.ndarray]*, *pi: Optional[Union[str, numpy.ndarray, list]] = None*, *end: Optional[Union[str, numpy.ndarray, list]] = None*, *seed: Optional[int] = None*) → object

Analyze sequences of observations and states.

**Parameters**

- **obs_seq** (*Union[str, list, numpy.ndarray]*) – Sequence of observations (of size N). Observation space, O = [o_1, o_2, …, o_N].
- **states_seq** (*Union[str, list, numpy.ndarray]*) – Sequence of states (of size K). State space = [s_1, s_2, …, s_K].
- **pi** (*Optional[Union[str, list, numpy.ndarray]]*) – Initial state probabilities array (of size K). If None, array is sampled from a uniform distribution.
- **end** (*Optional[Union[str, list, numpy.ndarray]]*) – Initial state probabilities array (of size K). If None, array is sampled from a uniform distribution.

- **seed** (*Optional[int]*) – Random state used to draw random variates. Passed to *scipy.stats.uniform* method.

**Returns** **model** – Hidden Markov model learned from the given data.

**Return type** *HiddenMarkovModel*

**graph_make**(**args*, ***kwargs*) → graphviz.dot.Digraph

Make a directed graph of a Hidden Markov model using *graphviz*.

**Parameters**

- **args** (*optional*) – Arguments passed to the underlying *graphviz.Digraph* method.
- **kwargs** (*optional*) – Keyword arguments passed to the underlying *graphviz.Digraph* method.

**Returns** **graph** – Digraph object with its own methods.

**Return type** graphviz.dot.Digraph

**Note:** *graphviz.dot.Digraph.render* method should be used to output a file.

**viterbi**(*obs_seq: Union[str, list, numpy.ndarray]*, *obs: Optional[Union[list, numpy.ndarray]] = None*, *states: Optional[Union[list, numpy.ndarray]] = None*, *tp: Optional[numpy.ndarray] = None*, *ep: Optional[numpy.ndarray] = None*, *pi: Optional[Union[list, numpy.ndarray]] = None*) → Tuple[numpy.ndarray, numpy.ndarray]

Viterbi algorithm.

**Parameters**

- **obs_seq** (*Union[str, list, np.ndarray]*) – Sequence of observations.
- **obs** (*Optional[Union[list, np.ndarray]]*) – Observations space (of size N).
- **states** (*Optional[Union[list, np.ndarray]]*) – List of states (of size K).
- **tp** (*Optional[numpy.ndarray]*) – Transition matrix (of size K × K) which stores transition probability of transiting from state i (row) to state j (col).
- **ep** (*Optional[numpy.ndarray]*) – Emission matrix (of size K × N) which stores probability of seeing observation j (col) from state i (row). N is the length of observation space, O = [o_1, o_2, …, o_N].
- **pi** (*Optional[Union[list, np.ndarray]]*) – Initial probabilities array (of size K).

**Returns**

- **x** (*numpy.ndarray*) – Sequence of states.
- **z** (*numpy.ndarray*) – Sequence of state indices.

# INDEX

## Symbols



## C



## F



## G



## H



## M



## N



## P



## S



## V