ChainoPy: A Python Library for Discrete Time Markov Chains and Markov Chain Neural Networks

Statement of Need

There are significant limitations in current Markov Chain packages that rely solely on NumPy (Harris et al. 2020) and Python for implementation. Markov Chains often require iterative convergence-based algorithms (Rosenthal 1995), where Python's dynamic typing, Global Interpreter Lock (GIL), and garbage collection can hinder potential performance improvements like parallelism. To address these issues, we enhance our library with extensions like Cython and Numba for efficient algorithm implementation. Additionally, we introduce a Markov Chain Neural Network (Awiszus and Rosenhahn 2018) that simulates given Markov Chains while preserving statistical properties from the training data. This approach eliminates the need for post-processing steps such as sampling from the outcome distribution while giving neural networks stochastic properties rather than deterministic behavior. Finally, we implement the famous Markov Switching Models (Hamilton 2010) which are one of the fundamental and widely used models in Finance applications such as Stock Market price prediction.

Implementation

We implement three public classes `MarkovChain`, `MarkovChainNeuralNetwork` and `MarkovSwitchingModel` that contain core functionalities of the package. Performance intensive functions for the `MarkovChain` class are implemented in the `_backend` directory where a custom cython (Behnel et al. 2010) backend is implemented circumventing drawbacks of python like the GIL, dynamic typing etc. The `MarkovChain` class implements various functionalities for discrete-time Markov chains. It provides methods for fitting the transition matrix from data, simulating the chain, calculating properties such as ergodicity, irreducibility, symmetry, and periodicity, as well as computing stationary distributions, absorption probabilities, expected time to absorption, and expected number of visits. It also supports visualization of the transition matrix and chain.

We do the following key optimizations:

- Efficient matrix power: If the matrix is diagonalizable, an eigenvalue decomposition based matrix power is performed.
- Parallel Execution: Some functions are parallelized.
- JIT compilation with Numba (Lam, Pitrou, and Seibert 2015): Numba is used for just-in-time compilation to improve performance.
- `__slots__` usage: `__slots__` is used instead of `__dict__` for storing object attributes, reducing memory overhead.
- Caching decorator: Class methods are decorated with caching to avoid recomputation of unnecessary results.
- Direct LAPACK use: LAPACK function `dgeev` is directly used to calculate stationary-distribution via SciPy's (Virtanen et al. 2020) `cython_lapack` API instead of additional numpy overhead.
- Utility functions for visualization: Utility functions are implemented for visualizing the Markov chain.
- Sparse storage of transition matrix: The model is stored as a JSON object, and if 40% or more elements of the transition matrix are near zero, it is stored in a sparse format.

The `MarkovChainNeuralNetwork` implementation defines a neural network model, MarkovChainNeuralNetwork, using PyTorch (Paszke et al. 2019) for simulating Markov chain behavior. It takes a Markov chain object and the number of layers as input, with each layer being a linear layer. The model's forward method computes the output probabilities for the next state. The model is trained using stochastic gradient descent (SGD) with a learning rate scheduler. Finally, the model's performance is evaluated using the KL divergence between the original Markov chain's transition probabilities and those estimated from the simulated walks.

The steps to generate training data as described in (Awiszus and Rosenhahn 2018) are as follows:

```
1. Input Data Augmentation: Add a random value (r) between 0 and 1
to the input data. This value influences the output, simulating the
Markov chain's probabilistic nature.

2. Cumulative Frequency Calculation: Calculate the cumulative
frequency for each possible transition from the current state to
the next states based on transition probabilities.

3. Training Data Generation: Generate training data by sampling
random numbers (r) and selecting the next state based on the
calculated cumulative frequencies. This reflects the Markov chain's
transition probabilities.

Example: If the transition probabilities from state 1 to states 2, 3,
and 4 are 1/3 each, the cumulative frequencies would be [0, 1/3, 2/3, 1].
For instance, with a random number of 0.5, the next state might be 3,
```

resulting in the pair (0.5, 1, 0, 0, 0) → (0, 0, 1, 0) for state 1.

API of the library:

- chainopy.MarkovChain(transition-matrix: ndarray, states: list)

```
        Public Methods
        --------------

        - fit()
        - simulate()
        - predict()
        - adjacency_matrix()
        - nstep_distribution()
        - is_ergodic()
        - is_symmetric()
        - stationary_dist()
        - is_absorbing()
        - is_aperiodic()
        - period()
        - is_irreducible()
        - is_transient(state)
        - is_recurrent(state)
        - fundamental_matrix()
        - absorption_probabilities()
        - expected_time_to_absorption()
        - expected_number_of_visits()
        - expected_hitting_time(state)
        - visualize_transition_matrix()
        - visualize_chain()
        - save_model()
        - load_model()
        - marginal_dist()
        - fit_from_file()
```

- chainopy.MarkovChainNeuralNetwork(chainopy.MarkovChain, num_layers)

```
        Public Methods
        --------------

        - train_model()
        - get_weights()
        - simulate_random_walk()
```

- chainopy.MarkovSwitchingModel()

```
        Public Methods
```

```
--------------

- fit()
- predict()
- evaluate()
```

```
- chainopy.divergance_analysis(MarkovChain, MarkovChainNeuralNetwork)
```

Documentation, Testing and Benchmarking

For Documentation we use Sphinx. For Testing and Benchmarking the `MarkovChain` class we use the Pytest and PyDTMC ("PyDTMC," n.d.) package.

The results are as follows:

- `is_absorbing` Methods

Transition-Matrix Size	10	50	100	500	1000	2500
	Mean	St. dev	Mean	St. dev	Mean	St. dev
Function						
1. is_absorbing (ChainoPy)	97.3ns	2.46ns	91.8ns	0.329ns	98ns	0.4ns
1. is_absorbing (PyDTMC)	386ns	5.79ns	402ns	2.01ns	417ns	3ns

- `stationary_dist` vs `pi` Methods

Transition-Matrix Size	10	50	100	500	1000	2500
	Mean	St. dev	Mean	St. dev	Mean	St. dev
Function						
1. stationary_dist (ChainoPy)	1.47us	1.36us	93.4ns	5.26ns	96.6ns	3.9ns
1. pi (PyDTMC)	137us	12.9us	395ns	15.4ns	398ns	10.5ns

- `fit` vs `fit_sequence` Method:

Number of Words	10	50	100	500	1000	2500
	Mean	St. dev	Mean	St. dev	Mean	St. dev
Function						
1. fit (ChainoPy)	116 µs	5.28 µs	266 µs	15 µs	496 µs	47.3 µs
1. fit_sequence (PyDTMC)	14 ms	1.74 ms	14.4 ms	1.17 ms	17.3 ms	2.18 ms

- `simulate` Method

Transition-Matrix Size	N-Steps	ChainoPy Mean	ChainoPy St. dev	PyDTMC Mean	PyDTMC St. dev
10	1000	22.8 ms	2.32 ms	28.2 ms	933 µs
	5000	86.8 ms	2.76 ms	155 ms	5.25 ms
50	1000	17.6 ms	1.2 ms	29.9 ms	1.09 ms
	5000	84.5 ms	4.84 ms	161 ms	7.62 ms
100	1000	21.6 ms	901 µs	37.4 ms	3.99 ms
	5000	110 ms	11.3 ms	162 ms	5.75 ms
500	1000	24 ms	3.73 ms	39.6 ms	6.07 ms
	5000	112 ms	6.63 ms	178 ms	26.5 ms
1000	1000	26.1 ms	620 µs	46.1 ms	6.47 ms
	5000	136 ms	2.49 ms	188 ms	2.43 ms
2500	1000	42 ms	3.77 ms	59.6 ms	2.29 ms
	5000	209 ms	16.4 ms	285 ms	27.6ms

Apart from this, we test the `MarkovChainNeuralNetworks` by training them and comparing random walks between the original `MarkovChain` (Right) object and those generated by `MarkovChainNeuralNetworks` (Left) through a Histogram.

The results for a 2 x 2 Markov Chain are as follows:

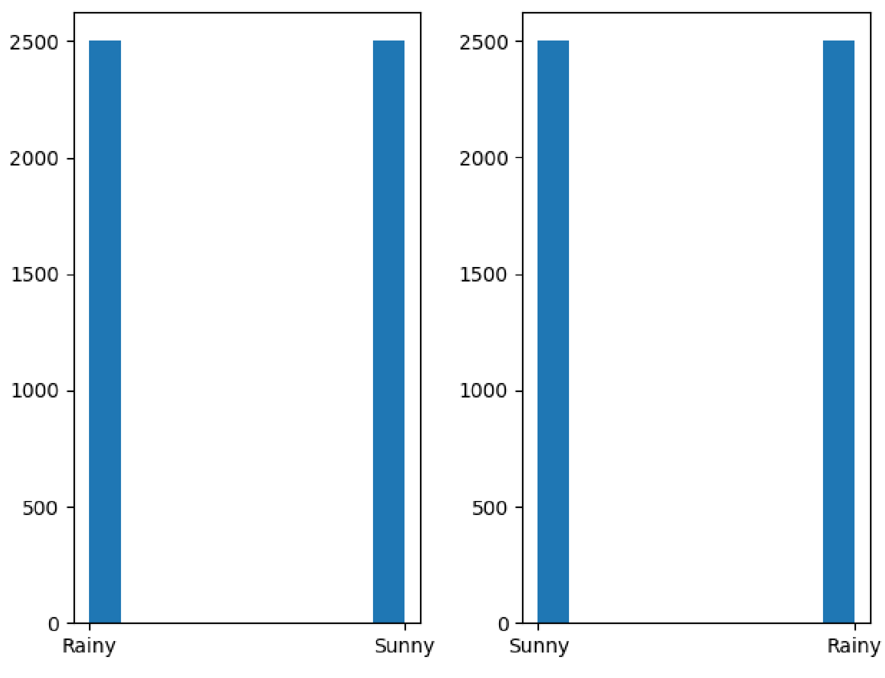

Figure 1: 2 x 2 Simulation

The results for a 3 x 3 Markov Chain are as follows:

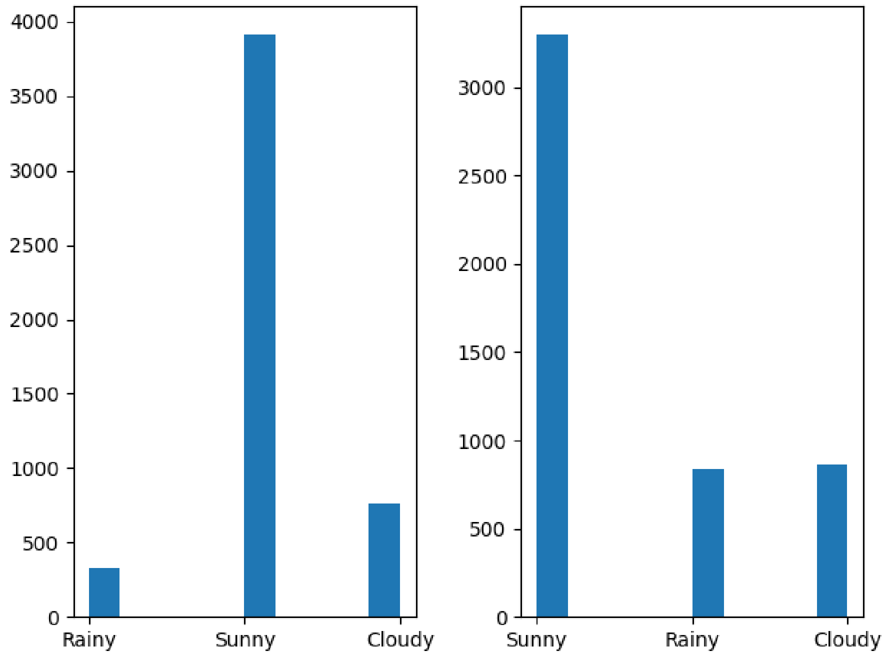

Figure 2: 3 x 3 Simulation

Conclusion

In conclusion, ChainoPy offers a Python library for discrete-time Markov Chains and includes features for Markov Chain Neural Networks, providing a useful tool for researchers and practitioners in stochastic analysis with efficient performance.

References

Awiszus, Maren, and Bodo Rosenhahn. 2018. "Markov Chain Neural Networks." In *Proceedings of the Ieee Conference on Computer Vision and Pattern Recognition Workshops*, 2180–7.

Behnel, Stefan, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2010. "Cython: The Best of Both Worlds." *Computing in Science & Engineering* 13 (2): 31–39.

Hamilton, James D. 2010. "Regime Switching Models." In *Macroeconometrics and Time Series Analysis*, 202–9. Springer.

Harris, Charles R, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, et al. 2020. "Array Programming with Numpy." *Nature* 585 (7825): 357–62.

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert. 2015. "Numba: A Llvm-Based Python Jit Compiler." In *Proceedings of the Second Workshop on the Llvm Compiler Infrastructure in Hpc*, 1–6.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. "Pytorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in Neural Information Processing Systems* 32.

"PyDTMC." n.d. https://github.com/TommasoBelluzzo/PyDTMC.

Rosenthal, Jeffrey S. 1995. "Convergence Rates for Markov Chains." *Siam Review* 37 (3): 387–405.

Virtanen, Pauli, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, et al. 2020. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." *Nature Methods* 17 (3): 261–72.