`markovml`: A Python Package for Verifying Markov Processes with Embedded Machine Learning Models

Muhammad Maaz [1] **Timothy C. Y. Chan** [1]

Abstract

We present `markovml`, a Python package for verifying properties of Markov processes whose parameters are defined by machine learning (ML) models. While existing tools support formal verification of either Markov processes or ML models individually, none support reasoning about systems that integrate both. `markovml` fills this gap by allowing users to construct Markov processes and embed pretrained models from standard ML libraries, including linear models, decision trees, and neural networks. The package provides an expressive domain-specific language to specify constraints and verify common properties such as reachability, expected hitting time, and total reward. We illustrate usage through several examples and release the package with full documentation and tutorials. Our package is available at `https://github.com/mmaaz-git/markovml`.

1. Introduction

Markov processes are fundamental mathematical models used across computer science, operations research, engineering, and healthcare (Stewart, 2021). In computer science, they capture the behavior of probabilistic systems such as hardware, communication protocols, or autonomous agents (Baier & Katoen, 2008). In engineering, they are used to analyze degradation and failure in complex machinery (Rausand & Hoyland, 2003). In healthcare, they model patient transitions between clinical states and underpin cost-effectiveness analyses of medical interventions (Sonnenberg & Beck, 1993).

As machine learning (ML) becomes increasingly integrated into real-world systems, Markov models are evolving to incorporate heterogeneous, data-driven parameters. For instance, ML models can estimate failure rates from sensor data or patient-specific transition probabilities based on clinical features (Mertens et al., 2022). This gives rise to Markov processes where parameters are *learned functions*.

Despite this growing trend, there is limited work on rigorously analyzing such systems. In healthcare, most studies rely on Monte Carlo simulation (Krijkamp et al., 2018), but cannot provide formal guarantees. While our primary motivation is healthcare, the approach generalizes to any application where learned models feed into Markov processes. This enables us to rigorously answer questions such as: Given a bound on the input, what is the worst-case probability of reaching a failure state? Is the failure rate for machines with certain properties guaranteed to remain below 0.01%? For a clinical subgroup, is the expected treatment cost within a government threshold?

In this paper, we present our open-source software package, `markovml`, for the formal verification of Markov processes with ML-based parameters, available at `https://github.com/mmaaz-git/markovml`. The technical and mathematical details, as well as our novel algorithm for solving the resulting problems orders of magnitude faster than existing solvers, are in a companion paper (Maaz & Chan, 2025). In this paper, we focus on discussing the software itself, how users can interact with it, and how it interfaces with other ML libraries.

Related Software There exist several tools for *probabilistic model checking*, which verifies properties of systems modeled by Markov chains (Hansson & Jonsson, 1994; Baier & Katoen, 2008), e.g., PRISM and others (Kwiatkowska et al., 2002; Katoen et al., 2005; Hermanns et al., 2000; Sen et al., 2005; Younes, 2005; Lassaigne & Peyronnet, 2002). Concurrently, there also exist several tools for verifying machine learning models, which take a pretrained model and obtain guaranteed bounds on outputs, e.g., α, β-CROWN (Wang et al., 2021; Zhang et al., 2022b;a; Kotha et al., 2023), VeriNet (Henriksen & Lomuscio, 2020; 2021), and Marabou (Katz et al., 2019; Wu et al., 2020). In the healthcare domain, simulation software like TreeAge (**?**) allow for heterogeneous parameters of the Markov chain

[*] Equal contribution [1] University of Toronto. Correspondence to: Muhammad Maaz <m.maaz@mail.utoronto.ca>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25). Copyright 2025 by the author(s).

drawn from distributions, but currently only supports Monte Carlo simulation. Our software is thus the first tool that allows the formal verification of integrated Markov models and ML models, and, by formulating the problem as an optimization problem and solving it to global optimality, it provides robust guarantees that simulation cannot.

2. Mathematical Background

Notation Vectors are lowercase bold, e.g., $\mathbf{x}$, with i-th entry x_i, and matrices by uppercase bold letters, e.g., $\mathbf{M}$ with (i, j)-th entry M_{ij}. The identity matrix is denoted by $\mathbf{I}$, the vector of all ones by $\mathbf{1}$, with dimensions inferred from context. The set of integers from 1 to n is denoted by $[n]$.

Preliminaries A (discrete-time, finite-state) *Markov chain* with n states, indexed by $[n]$, is defined by a row-stochastic transition matrix $\mathbf{P} \in \mathbb{R}^{n\times n}$, where P_{ij} is the probability of transitioning from state i to state j, and a stochastic initial distribution vector $\boldsymbol{\pi} \in \mathbb{R}^n$, where π_i is the probability of starting in state i. Furthermore, if we assign rewards to each state, we call this a *Markov reward process*. A Markov reward process has a reward vector $\mathbf{r} \in \mathbb{R}^n$, where r_i is the reward for being in state i for one period. A state is *absorbing* if it cannot transition to any other state, and *transient* otherwise.

Below, we recount three of the key properties commonly computed in practice (Puterman, 2014).

Definition 2.1 (Reachability)**.** The probability of eventually reaching a set of states $S \subseteq [n]$, from a set $T \subseteq [n]$ of transient states equal to the complement of S, assuming that the chain will reach S from T with probability 1, is given by $\tilde{\boldsymbol{\pi}}^\top(\mathbf{I} - \mathbf{Q})^{-1}\mathbf{R}\mathbf{1}$, where $\mathbf{Q}$ is the transition matrix restricted to T, $\mathbf{R}$ is the transition matrix from T to S, and $\tilde{\boldsymbol{\pi}}$ is the initial distribution over T.

Definition 2.2 (Expected hitting time)**.** The expected number of steps to eventually reach a set of states $S \subseteq [n]$, from a set $T \subseteq [n]$ of transient states equal to the complement of S, assuming that the chain will reach S from T with probability 1, is given by $\tilde{\boldsymbol{\pi}}^\top(\mathbf{I} - \mathbf{Q})^{-1}\mathbf{1}$, where $\mathbf{Q}$ is the transition matrix restricted to T, and $\tilde{\boldsymbol{\pi}}$ is the initial distribution over T.

Definition 2.3 (Total infinite-horizon discounted reward)**.** The total infinite-horizon discounted reward, with a discount factor $\lambda \in (0, 1)$ is given by $\sum_{t=0}^{\infty} \lambda^t \boldsymbol{\pi}^\top \mathbf{P}^t \mathbf{r} = \boldsymbol{\pi}^\top (\mathbf{I} - \lambda\mathbf{P})^{-1} \mathbf{r}$.

We wish to *verify* these properties, namely finding their maximum or minimum, which we will do by formulating and solving an optimization problem.

Embedding ML Models We now consider $\boldsymbol{\pi}, \mathbf{P}, \mathbf{r}$ as functions of a *feature vector* $\mathbf{x} \in \mathbb{R}^m$, where m is the number of features. Given one or more ML models, we can write elements of $\boldsymbol{\pi}, \mathbf{P}, \mathbf{r}$ as affine functions of the outputs of the ML models. We also allow additional linear inequalities on the elements of $\boldsymbol{\pi}, \mathbf{P}, \mathbf{r}$. Lastly, $\mathbf{x}$ is constrained to lie in a set $\mathcal{X}$.

Our key assumption is that the ML models are *mixed-integer linear representable* (MILP-representable), meaning that the relationship between the inputs and outputs can be expressed using linear constraints and binary variables. This is a broad class of functions that includes, e.g., piecewise linear functions (including simple "if-then" rules), linear and logistic regression, tree-based models, and neural networks with ReLU activations (we handle the non-linearity introduced by softmax or logistic with spatial branch-and-bound). We also assume that $\mathcal{X}$ is a MILP-representable set (Jeroslow & Lowe, 1984). See Figure 1 for the full pipeline.

Solving the problem As discussed in greater detail in the companion paper (Maaz & Chan, 2025), under these assumptions, verifying the three aforementioned properties can be formulated as a *bilinear program*, an NP-hard nonconvex quadratic program. We developed a novel decomposition scheme which significantly speeds up the solution of the resulting bilinear program. This algorithm is fully implemented in our software.

3. Overview of `markovml`

`markovml`, allows the user to specify Markov chains or reward processes with embedded pretrained machine learning models. Our domain-specific language lets users: (1) instantiate a Markov process, (2) add pretrained ML models from `sklearn` (Pedregosa et al., 2011) and PyTorch (Paszke et al., 2019), (3) link model outputs to Markov parameters with affine equalities, (4) include extra linear inequalities, (5) specify the feature set with MILP constraints, and (6) optimize reachability, hitting time, or total reward.

Our package is built on the optimization solver Gurobi, specifically its Python interface, `gurobipy` (Gurobi Optimization, LLC, 2024). As the user sets up their model, our software constructs, in the background, the equivalent formulation in Gurobi. We also then directly leverage Gurobi's ability to solve bilinear programs to global optimality (our decomposition scheme also calls out to Gurobi's solver).

Supported models Our package supports a variety of regression and classification models, including linear, tree-based, and neural networks. For some models, the MILP formulation is provided by `gurobi-machinelearning` (Gurobi Optimization, 2024), a Python package built on top of `gurobipy` which embeds pretrained ML models into a Gurobi model, while for others, we implemented the MILP formulation ourselves.

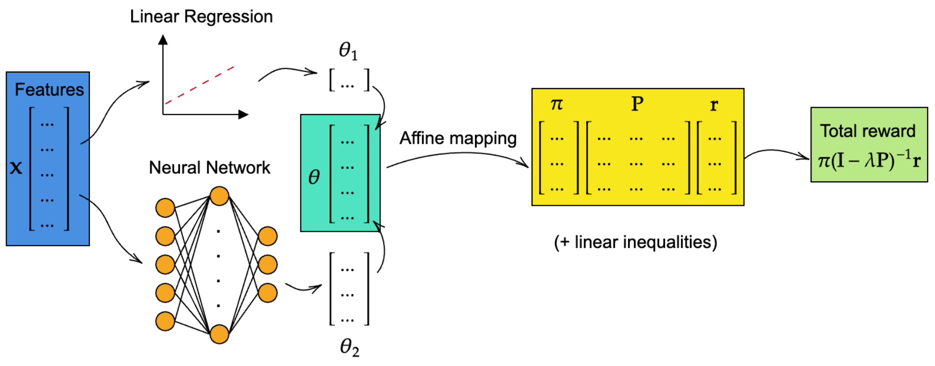

Figure 1. Example of our pipeline. A feature vector $\mathbf{x}$ is passed through different functions, here a linear regression and a neural network, to obtain the output vector $\boldsymbol{\theta}$, which then determines the parameters of the Markov process through affine equalities.

We support the following models from `sklearn`:

- `LinearRegression`
- `Ridge`
- `Lasso`
- `LogisticRegression`
- `DecisionTreeRegressor`
- `DecisionTreeClassifier`
- `RandomForestRegressor`
- `RandomForestClassifier`
- `MLPRegressor`
- `MLPClassifier`

From `pytorch`, we support neural networks built as `nn.Sequential` models with ReLu or linear layers, with possibly a softmax layer at the end for classifiers.

Lastly, we implemented a model called `DecisionRules`, which allows the user to specify a series of "if-then" rules specified in natural language, e.g., "if age $\geq$ 65 then 0.8". This enables encoding tables from the literature, like age-stratified mortalities, common in healthcare.

4. Using markovml

We walk through how to use `markovml`. Our software comes bundled with extensive documentation and tutorials.

Instantiating a Markov process There are three objects in the `markovml` package: `MarkovReward`, `MarkovReach`, and `MarkovHitting`. They correspond to the problem you are trying to solve: verifying the total reward, reachability, and hitting time, respectively. They have slightly different interfaces and internal operations, but there is lots of common functionality, so it is easy to switch between them (in fact, they all inherit from the same base class, `AbstractMarkov`, which provides almost all of the functionality). While they have common setups, there are specific elements to each of them; for example, `MarkovReward` requires the setting of a reward vector $\mathbf{r}$ while the others do not. Below are some examples of instantiating each of these objects.

```
mrp = MarkovReward(n_states=2, n_features
    =2)
m_reach = MarkovReach(n_states=2,
    n_features=2, n_transient=1, n_targets
    =1)
m_hitting = MarkovHitting(n_states=2,
    n_features=2, n_transient=1)
```

Adding ML models The next step is to add a machine learning model to the problem. This is done with the `add_ml_model` function. This function takes a pretrained model as input and embeds its MILP formulation in the underlying optimization problem. Suppose we have pretrained models `reward_model`, `trans_model`, `reward_model2`, `trans_model2`, where the reward models are regression models and the transition models are classification models. We can add them to the underlying Markov process as:

```
mrp.add_ml_model(reward_model)
mrp.add_ml_model(trans_model)
mrp.add_ml_model(reward_model2)
mrp.add_ml_model(trans_model2)
```

We can then access the variables corresponding to the *outputs* of each of these models using the `ml_outputs` attribute of the Markov object; e.g., `mrp.ml_outputs[0][0]` is the first output of the first added model (`reward_model`).

Setting parameters There are three ways to set the parameters of the Markov process: (1) pass constants at initialization, (2) use `set_to_const` or `set_to_ml_output`, or (3) use the setting helper functions.

If it is known that an entire vector or matrix is a constant, then this can be passed at initialization, as below.

```
MarkovReward(n_states=2, n_features=2, r
    =[1, 0])
```

The set_to_const or set_to_ml_output functions can be used to set elements one at a time either to a constant or to an affine function of one of the ML models' outputs, as below.

```
mrp.set_to_const(mrp.r[0], 1)
mrp.set_to_const(mrp.r[1], 0)

mrp.set_to_ml_output(mrp.r[2], mrp.
    ml_outputs[0][0])
mrp.set_to_ml_output(mrp.r[3], 2*mrp.
    ml_outputs[0][0]-1)
```

Lastly, each of the classes have setting helper functions which can be used to set the parameters altogether, as it may be cumbersome to set one at a time. Depending on the class, these are: set_pi, set_P, set_Q, set_r, set_R.

```
mrp.set_pi([mrp.ml_outputs[1][0],
            1 - mrp.ml_outputs[1][0],
            0, 0, 0])
```

Adding linear constraints on parameters We can add linear constraints on the parameters quite simply:

```
mrp.add_constraint(mrp.r[0] >= 1)
```

Defining the feature space To define $\mathcal{X}$, we use add_feature_constraint and add_feature_aux_variable. The first function allows adding linear inequalities, and the second allows adding auxiliary continuous or discrete variables. Together, these can encode any MILP-representable set.

```
mrp.add_feature_constraint(mrp.features[0]
    + mrp.features[1] <= 1.5)
mrp.add_feature_constraint(mrp.features[2]
    >= mrp.features[3])
```

Optimizing Once all parameters have been set and the feature space has been fixed, we can optimize our metric of interest quite simply.

```
mrp.optimize(sense="max")
```

It is possible to pass various solving options here as well as find a feasible solution instead of optimizing.

Complete example In only a few lines, a user can build a Markov process, integrate an ML model, and optimize the reward.

```
from markovml.markovml import MarkovReward
from sklearn.linear_model import
    LogisticRegression
import numpy as np

# Create a Markov reward process
mrp = MarkovReward(n_states=2, n_features
    =2)

# fix some parameters
mrp.set_r([1, 0])
mrp.set_pi([1, 0])

# train a classifier
X = np.random.rand(100, 2)
y = np.random.randint(0, 2, 100)
clf = LogisticRegression().fit(X, y)

# add classifier
mrp.add_ml_model(clf)

# link to transitions
mrp.set_P([[1 - mrp.ml_outputs[0][0], mrp.
    ml_outputs[0][0]], [0, 1]])

# define feature space
mrp.add_feature_constraint(mrp.features[0]
    >= 65)
mrp.add_feature_constraint(mrp.features[1]
    >= 100)

# optimize!
mrp.optimize(sense="max")
```

Decision rules The following code demonstrates building a DecisionRules model, which allows building "if-then" rules in a natural language syntax.

```
dr = DecisionRules(features = ['age', '
    income'])
dr.fit(rules = [
     "if age > 20 then 2.5",
     "if income >= 50000 and age < 30 then
    -1.0",
     "else 0.0"
])
```

It is then possible to "predict" using such a model on new data. This can also be added into a Markov object, just as any other learned model, and we implemented the MILP formulation through a series of logical constraints.

5. Conclusion

markovml bridges a critical gap in formal verification tools by enabling the verification of Markov processes whose parameters are given by ML models. By offering a simple and expressive syntax, it makes formal verification accessible for real-world, data-driven applications across domains like healthcare, reliability, and decision making.

References

Baier, C. and Katoen, J.-P. *Principles of model checking*. MIT press, 2008.

Gurobi Optimization, L. Gurobi machine learning, 2024. URL https://pypi.org/project/gurobi-machinelearning/. Python package for integrating regression models into optimization problems.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL https://www.gurobi.com.

Hansson, H. and Jonsson, B. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6: 512–535, 1994.

Henriksen, P. and Lomuscio, A. Efficient neural network verification via adaptive refinement and adversarial search. In *ECAI 2020*, pp. 2513–2520. IOS Press, 2020.

Henriksen, P. and Lomuscio, A. Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis. In *IJCAI*, pp. 2549–2555, 2021.

Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., and Siegle, M. A markov chain model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 347–362. Springer, 2000.

Jeroslow, R. G. and Lowe, J. K. *Modelling with integer variables*, pp. 167–184. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

Katoen, J.-P., Khattri, M., and Zapreevt, I. A markov reward model checker. In *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pp. 243–244. IEEE, 2005.

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*, pp. 443–452. Springer, 2019.

Kotha, S., Brix, C., Kolter, J. Z., Dvijotham, K., and Zhang, H. Provably bounding neural network preimages. In Oh, A., Neumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 80270–80290, 2023.

Krijkamp, E. M., Alarid-Escudero, F., Enns, E. A., Jalal, H. J., Hunink, M. M., and Pechlivanoglou, P. Microsimulation modeling for health decision sciences using r: a tutorial. *Medical Decision Making*, 38(3):400–422, 2018.

Kwiatkowska, M., Norman, G., and Parker, D. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 200–204. Springer, 2002.

Lassaigne, R. and Peyronnet, S. Approximate verification of probabilistic systems. In *Joint International Workshop von Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pp. 213–214. Springer, 2002.

Maaz, M. and Chan, T. C. Formal verification of markov processes with learned parameters. *arXiv preprint arXiv:2501.15767*, 2025.

Mertens, E., Genbrugge, E., Ocira, J., and Peñalvo, J. L. Microsimulation modeling in food policy: A scoping review of methodological aspects. *Advances in Nutrition*, 13(2):621–632, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Puterman, M. L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Rausand, M. and Hoyland, A. *System reliability theory: models, statistical methods, and applications*, volume 396. John Wiley & Sons, 2003.

Sen, K., Viswanathan, M., and Agha, G. On statistical model checking of stochastic systems. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*, pp. 266–280. Springer, 2005.

Sonnenberg, F. A. and Beck, J. R. Markov models in medical decision making: a practical guide. *Medical decision making*, 13(4):322–338, 1993.

Stewart, W. J. Introduction to the numerical solution of markov chains. 2021.

Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.-J., and Kolter, J. Z. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *Advances in Neural Information Processing Systems*, 34, 2021.

Wu, H., Ozdemir, A., Zeljic, A., Julian, K., Irfan, A., Gopinath, D., Fouladi, S., Katz, G., Pasareanu, C., and Barrett, C. Parallelization techniques for verifying neural networks. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, volume 1, pp. 128–137. TU Wien Academic Press, 2020.

Younes, H. L. Ymer: A statistical model checker. In *International Conference on Computer Aided Verification*, pp. 429–433. Springer, 2005.

Zhang, H., Wang, S., Xu, K., Li, L., Li, B., Jana, S., Hsieh, C.-J., and Kolter, J. Z. General cutting planes for bound-propagation-based neural network verification. *Advances in Neural Information Processing Systems*, 2022a.

Zhang, H., Wang, S., Xu, K., Wang, Y., Jana, S., Hsieh, C.-J., and Kolter, Z. A branch and bound framework for stronger adversarial attacks of ReLU networks. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pp. 26591–26604, 2022b.