Automated Rule learning for PRAM

Ashish Jha

1 Introduction

One of the core component of **PRAM** (Cohen and Loboda, 2019) Simulation Engine is **Rules** which forms the basis for redistribution of mass among groups. PRAM models have rules that are mutually exclusive condition and each condition is associated with a probability distribution. Currently, these rules have to be manually created based on modeler's judgment. Given the amount of historical data we have about events, we can also learn these rules by analyzing the statistical dependency between these sequential events. These events are partially stochastic in nature. We have developed a data-driven approach for extracting rules directly from observed data by exploiting these dependencies between them

2 Elements of automated rule generation

PRAM models are stochastic in nature and can successfully model many system's signals. Many of these systems can be represented using a Markov Model or Hidden Markov Model. Markov models are based on the *memory-less* property of stochastic process which states that conditional probability distribution of the future state of the process depends only upon the present state and not on the sequence of events that preceded it. This premise can be restrictive as there may be a dependency on the sequence of past events that leads to the future state. Thus we relax this property, by allowing a past sequence of events along with the current state to model the future state by determining dependency among them. We also minimize the complexity of the model by dropping states which are not statistically significant in the evolution of the future states.

Cohen and Oates (Cohen and Oates, 1998) developed **MSDD - Multiple stream dependency detection** algorithm for determining dependencies in frequently co-occurring patterns of values that occur in multiple streams of categorical data. These dependencies can expressed as rules of form: "If an instance of pattern x begins in the stream at time t then the instance of pattern y will occur at time $t+\delta$ with probability p. This algorithm finds strong dependency by performing a systematic search

over the space of all possible dependencies. It also explores the space of dependencies between pairs of arbitrary patterns of values and reduces both the left and right-hand side of the rules, which can be complex, to retain pattern which is frequently co-occurring. We build upon this algorithm to find statistically dependent relations between the sequence of events leading to future events

2.1 Rule generation for SIR flu Model

SIR flu model is a compartmental model developed by Kermack and McKendrick in 1927. This is a simple and well-understood model and the results can be easily evaluated. We have used our algorithm to learn rules about the spread of *flu* in the population. One of the key factors in the spread of flu is that it is proportional to the number of people who have it. Initially a very small proportion of population is *infected* with *flu* and rest of the population becomes *susceptible* to *flu*. Over the course of time a large number of population is *infected* but some proportion of *infected* population starts improving from the *flu* to reach status *recovered* status. Gradually most of the population starts improving from *flu* and reach status *recovered* and the model reaches a stationary distribution. This is a very simple model where the probability of transition to future state depends only upon the proportion of the population in the current state. We use data generated by the **PRAM** simulator based on an expert's rule to learn and reconstruct the same rule which was used to generate the data. This data comprises of transition of population mass among 3 states *viz* **[S,I,R]**

3 Evaluation and Analysis

For systematic evaluation and analysis of the rule generation module, we create a rule for the SIR model and run it through the simulation to generate the observation sequence **X**. This data is a string of symbols depicting the transition among the states at every time step. Then we run **X** through the learning algorithm to learn the sample transition parameters of the statistically significant features. The rule generation algorithm also generates an executable python code similar to the modeler's code [1] which we use to re-run the simulation again. We compare the model's parameters with the help of statistical tests and bar plot to compare generated parameters [1]. We also compare both models using standard metrics for performance evaluation:

Table 1: Code Comparison.

Listing 1: Actual Rule

```
class SIRRule(Rule):
  def apply(self, pop, group, iter, t):
    if group.has_attr({'flu': 'S'}):
      return [
        GroupSplitSpec(
          p=0.7,
          attr_set={'flu': 'S'}),
        GroupSplitSpec(
          p=0.3,
          attr_set={'flu': 'I'}),
      ]
    if group.has_attr({'flu': 'I'}):
      return [
        GroupSplitSpec(
          p=0.5,
          attr_set={'flu': 'I'}),
        GroupSplitSpec(
          p=0.5,
          attr_set={'flu': 'R'}),
      ]
    if group.has_attr({'flu': 'R'}):
      return [
        GroupSplitSpec(
          p=0.7,
          attr_set={'flu': 'R'}),
        GroupSplitSpec(
          p=0.3,
          attr_set={'flu': 'S'}),
      ]
```

Listing 2: Generated Rule

```
class Autogenerated1(Rule):
  def apply(self, pop, group, iter, t):
    if group.has_attr({'flu': 'S'}):
      return [
        GroupSplitSpec(
          p=0.68,
          attr_set={'flu': 'S'}),
        GroupSplitSpec(
          p=0.32,
          attr_set={'flu': 'I'}),
      ]
    if group.has_attr({'flu': 'I'}):
      return [
        GroupSplitSpec(
          p=0.52,
          attr_set={'flu': 'I'}),
        GroupSplitSpec(
          p=0.48,
          attr_set={'flu': 'R'}),
      ]
    if group.has_attr({'flu': 'R'}):
      return [
        GroupSplitSpec(
          p=0.69,
          attr_set={'flu': 'R'}),
        GroupSplitSpec(
          p=0.31,
          attr_set={'flu': 'S'}),
      ]
```

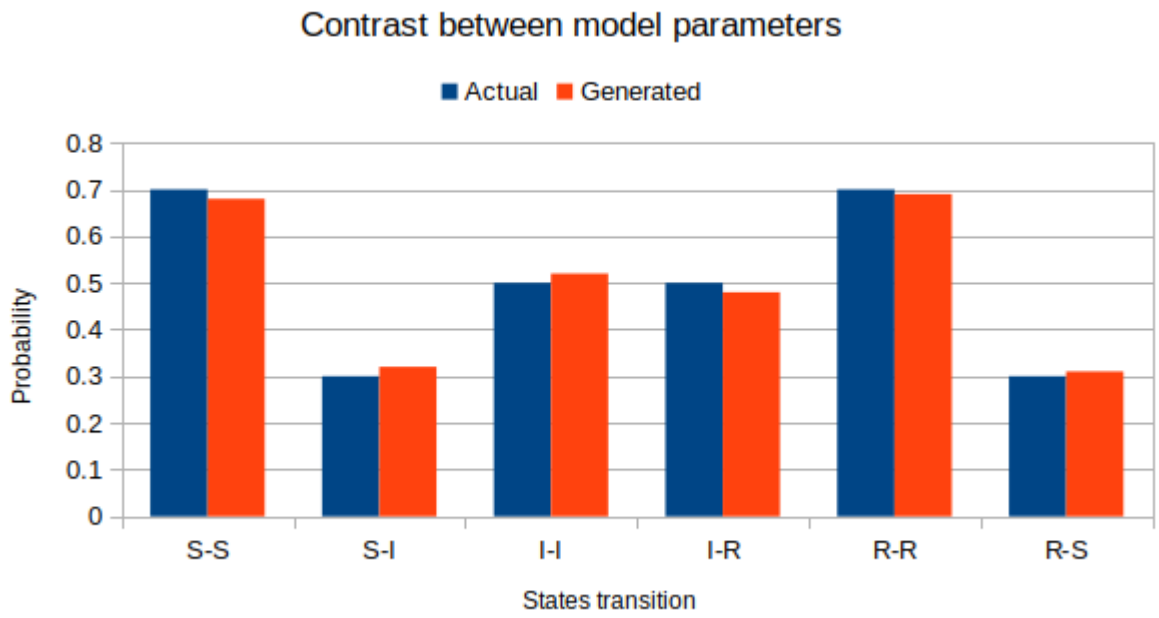

Figure 1: Model Parameters

3.1 Goodness of Fit among two models

If the algorithm learns from the data well then the distance between actual model parameters and learned model parameters should be small and unbiased. For this, we use **R-Squared** metrics to evaluate how well the generated parameters fit the actual model parameters. R-Squared is always between 0 and 1. 0 means the algorithm learns none of the variability of the actual model parameters. 1 indicates that the model explains all the variability of the model parameters. Generally, the higher the R-squared, the better the model. Table[2] below show the model parameters and the R-Squared score which is very close to 1. In conclusion, we can say that the generated model fits the actual parameters very well and they are identical.

Table 2: Model Parameters

	Actual	**Generated**
S-S	0.7	0.68
S-I	0.3	0.32
I-I	0.5	0.52
I-R	0.5	0.48
R-R	0.7	0.69
R-S	0.3	0.31

$$\text{R-Squared} = \frac{\text{Explained variation in model parameters}}{\text{Total variation in actual parameters}}$$
$$\text{R-Squared for the model} = 0.98875 \tag{1}$$

3.2 Divergence in steady state distribution of the two model

Due to the stochastic nature of this modeling stack, we also evaluate the steady-state distribution of both the models to evaluate how similar are generated distribution and actual distribution. We use **Kullback–Leibler divergence** which is the measure of relative entropy in the probability distribution of the two models. If the two distributions are identical then the Kullback–Leibler divergence between them is very close to 0. Figure 2 below shows the steady-state distribution of mass for both the models among groups. The distribution is identical and they also have very low divergence value in the equation [2]. Perhaps these two models produce similar steady-state distributions and are identical.

$$D_{KL}(p||q) = E[\log(p(x)) - \log(q(x))]$$
$$D_{KL}(Actual||Generated) = 0.00034 \tag{2}$$

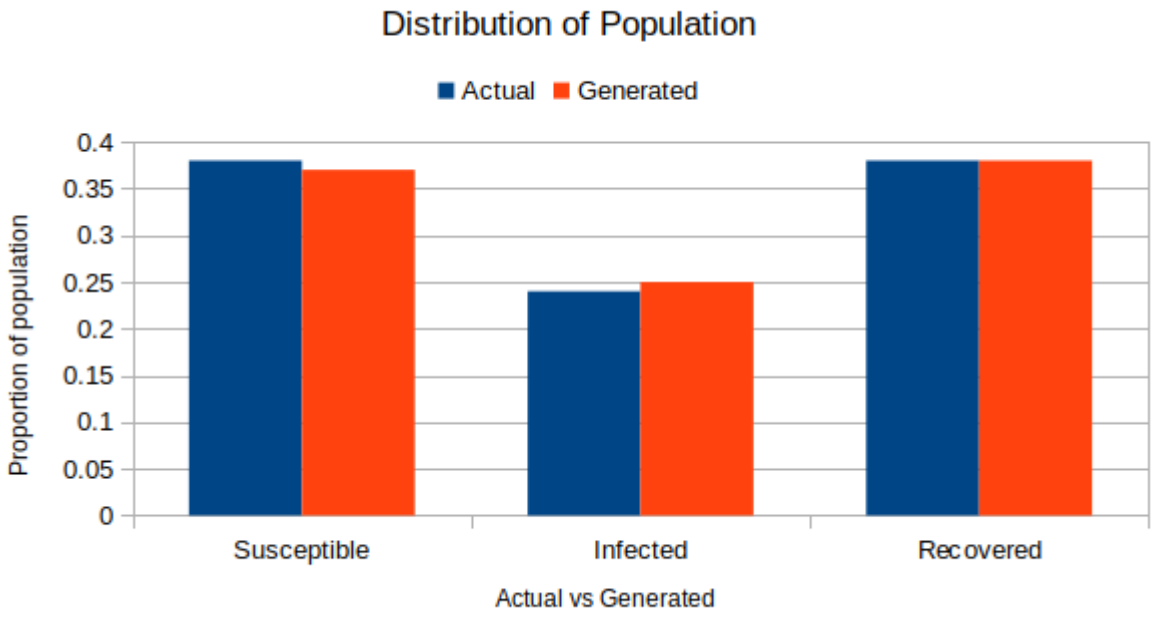

Figure 2: Steady state distribution

3.3 Statistical Test on model outputs

One of the key advantages of the simulation engine like **PRAM** is that we can learn the rule from data and use the generated rule directly in the simulator to generate the model output and compare it directly with the actual rule output. The simulation output is the count of the population in each group at every time step. For the evaluation and comparison of two simulation output we use the **Chi-square test (χ^2)**. The purpose of the Chi-square test is to determine whether the observed count of the population from the auto-generated rule differs from the count of population generated from the actual rule.

$$\chi^2 = \sum_{group} \frac{(Observed_{group} - Actual_{group})^2}{Actual_{group}} \tag{3}$$

3.3.1 Setup

- In our case of the SIR model we setup the simulation for 20-time steps with an initial population count of 1000 in susceptible **S** state.
- We set up the null Hypothesis H_0 and alternate hypothesis H_A.

  H_0 : The observed population in each category matches the actual population

  H_A : The observed population in each category does not match the actual population
- We run the simulation for both actual rule and auto-generated rule and then record the count

of population in each group at the end of simulation.

Table 3: Model Output

	Actual count	Generated count
S	384.6	369.9
I	230.8	247.1
R	384.6	383

- We run the Chi-square test on the population counts and compute the *p-value* with a significance level $\alpha = 0.05$

$$\chi^2 = 1.666$$
$$p - value = 0.434 \tag{4}$$

- Based on the resulting *p-value* we reject or accept the null hypothesis H_0

  Since the *p-value* is greater than significance level $\alpha = 0.05$ we fail to reject the H_0

- We can conclude that the population count in every group in both models is identical.

With the help of the Chi-squared test, we can conclude that indeed both the model produces identical population distribution among the groups and behave similarly. Thus the rule learned from data is similar to an actual rule created by the modeler

4 Conclusion

Rule learning will be a very useful tool to automatically generate models from data. This algorithm is useful for non-computer scientists with little knowledge of how the underlying system works and instead has just access to observations to construct a model that represents the system under study. We demonstrated a use case of rule extraction for **SIR** flu model and we were able to generate the parameters back from data generated by the actual rule. We also evaluated the performance of the rule generator algorithm which confirms that the algorithm can generate accurate models from historical data. We will expand the capabilities of the algorithm to learn multidimensional rules with even more convoluted relationships between them.

References

Cohen, Paul and Tim Oates. 1998. “Searching for Structure in Multiple Streams of Data.”.

Cohen, Paul and Tomek Loboda. 2019. “Redistribution Systems and PRAM.”.