National Research University Higher School of Economics

Faculty of Computer Science

Bachelor's Programme in Data Science and Business Analytics

TERM PAPER
Research Project
Mining process models from event logs using recurrent neural networks

Prepared by the student of Group DSBA192 in Year 3,
Gyursoy Aleksandr-Efe Mertovich

Term Paper Supervisor:
Candidate of Science, Professor, Sergei Andreevich Shershakov

Moscow
2022

Table of Contents

Abstract

Process mining is a relatively new research discipline, which have been recently gaining popularity. It sits somewhere between machine learning and data mining on the one hand and process modeling and analysis on the other. Process discovery task aims at building the processes in order to monitor, analyze and improve existing processes. In order to solve this task of building formal process models for their further use from an event logs a numerous methods and algorithms have been created and proposed over the past decades. In this paper we want to try a different method for process discovery using Recurrent Neural Networks. Neural Nets have proven their efficiency in capturing difficult non-linear dependencies between one object and another, whereas RNNs are capable of not only capturing dependencies but patterns and special behaviors in the data.

1 Introduction

Process models are a useful and powerful tool, existence of which allows for process automation and process evolution. Over the past few decades this sphere was gaining popularity more and more, and the process mining with the coming of big amounts of data have been becoming more and more viable. Due to the increasing complexity of processes within companies and the growing volume of data that needs to be processed and analyzed, it is becoming increasingly difficult to manually monitor inconsistencies and optimize the process. This is when the process mining become handy.

A process can be described as a set of events, which are taken step-by-step for an end to be attained. Most of the techniques that are applied to the process models require process models to exist in the first place. This led to the development of various techniques for process discovery. The essence of process discovery is to extract a model that reflects patterns and behaviors in the data, as a simplest example we may look at alpha-algorithm. In this paper we will focus on solving the task of process discovery using Recurrent Neural Networks (RNN).

The approach used in this paper is somewhat similar to the grammar inference, the language being the final process model and an event log being a sequence of sentences. RNN is going to be fed one event at a point of time and predict the next event in trace. The state of RNN may then be treated as a state in our process model. Our aim will be to construct a Transition System using the states of our RNN and an event log. The obtained model can then be refined to higher-level representation such as Petri Nets, which will allow to explicitly represent the concurrency. In this paper we will primarily look at the application of RNN and attempt to obtain a sensible process models using them.

2 Related Materials

Finite State Machines were attempted to be built in several other works. Some of them being very old works, when neural networks were a bit too hard to train thus leading to oversimplified models [1, 2]. Also, a recent attempt in inferring Transition Systems from event logs took place, using the Recurrent Neural Networks [3].

Study on activation functions and introduction of Gumbel-softmax activation function, showing to outperform other similar activation functions is presented in paper by Eric Jang, Shixiang Gu, Ben Poole [5].

3 Preliminaries

3.1 Process Mining concepts

We first want to define an event log. An event log consists of sequences of events. Event is an activity that was executed at a particular time and in particular case.

Definition 3.1.1 (Trace). *Let* $\boldsymbol{A}$ *be a set of activities of an event log. Then, trace* $\boldsymbol{t} \in \boldsymbol{A}^*$ *is a finite sequence of activities, where* $\boldsymbol{A}^*$ *is the set of all finite sequences over* $\boldsymbol{A}$.

Trace $\mathbf{t} = < a, b, b, d, e >$ is an example of a trace consisting of 5 activities. Then, an event log $\boldsymbol{L}$ is a collection of such traces.

Definition 3.1.2 (Event log). *An event log* $\boldsymbol{L}$ *is a finite multiset of observed traces*.

An event log is a multiset of different traces. As an example, lets take a look at an event log $\boldsymbol{L} = [< a, b, c, d, e >^{14}, < a, b, b, c >^{12}]$. It contains 26 traces overall, 14

of which are traces $< a, b, c, d, e >$ and the remaining 12 traces $< a, b, b, c >$.

Definition 3.1.3 (Transition System). *A transition system is a triplet* $\boldsymbol{TS = (S, A, T)}$*, where* $\boldsymbol{S}$ *is the set of states,* $\mathbf{A}$ *is the set of activities, and* $\boldsymbol{T \subset S \times A \times S}$ *is a set of transitions.* $\boldsymbol{S^{start} \subset S}$ *is a set of starting state, and* $\boldsymbol{S^{end} \subset S}$ *is a set of final states.*

3.2 Recurrent Neural Network

A Neural Network (NN) consists of three main part: a layer of input neurons, layers of hidden neurons and a layer of output neurons. Each layer consists of a number of neurons, each neuron accepting n inputs and producing a scalar output after applying an activation function. Also, each neuron is associated with some n-dimensional weight vector, which is being tuned during the training of the NN. At each step, the Neural Network is fed by some input and sequentially goes through all the layers defined in the architecture, thus producing an output.

In order to train the parameters present in the Neural Network loss function is introduced. The goal of the optimization is to find such parameters, that will either maximize of minimize the loss function depending on the case. This is done by changing the weight and bias vectors accordingly using the technique called back propagation. In simple words, back propagation is an algorithm, where we first evaluate the loss function, then evaluate partial derivative with respect to the weights of our Neural Network, combine individual gradients for each pair of input and output and combine them. At the last step we update the weights according to the learning rate and total gradient calculated for each weight.

In this work a Recurrent Neural Network (RNN) is going to be used. The main difference with NN is that there will be some of the information preserved from the previous steps and reused at each iteration. This will allow to capture the patterns in

the traces and make more sensible predictions in terms of given event logs.

We then define the activation functions to be used in Recurrent Neural Network, which are added in order to introduce non-linearity.

Definition 3.2.1 (Rectified Linear Unit). *ReLU is an activation function defined as* $f(x) = max(0, x)$*, where* x *is the input to a neuron.*

This function allows for a more efficient and faster training of Neural Network.

Definition 3.2.2 (Sigmoid). *Sigmoid activation function* $f(x) = \frac{1}{1+exp\,-x}$*, where* x *is the input to a neuron.*

Definition 3.2.3 (Softmax). $f(x_i) = \frac{e^{x_i}}{\sum e^{x_i}}$ *. In simple words, this activation functions reparametrizes each element of input vector* $\boldsymbol{x}$*, so that they represent the probability and the sum of* $f(x_i)$ *equals to 1.*

At last, a gumbel-softmax activation function is being used. It has proven to outperform existing stochastic gradient descent estimators and being effective at structured output learning. More details can be found in this paper.

Definition 3.2.4 (Categorical Cross-Entropy). Categorical Cross-Entropy is a loss function defined as follows: $CE = -\sum_i^C y_i * log(\hat{y}_i)$, where y_i is the corresponding target value (being 1 in ith position if it's the desired class and 0 otherwise) and $\hat{y}_i$ is the probability for the ith class.

4 Description of project approaches

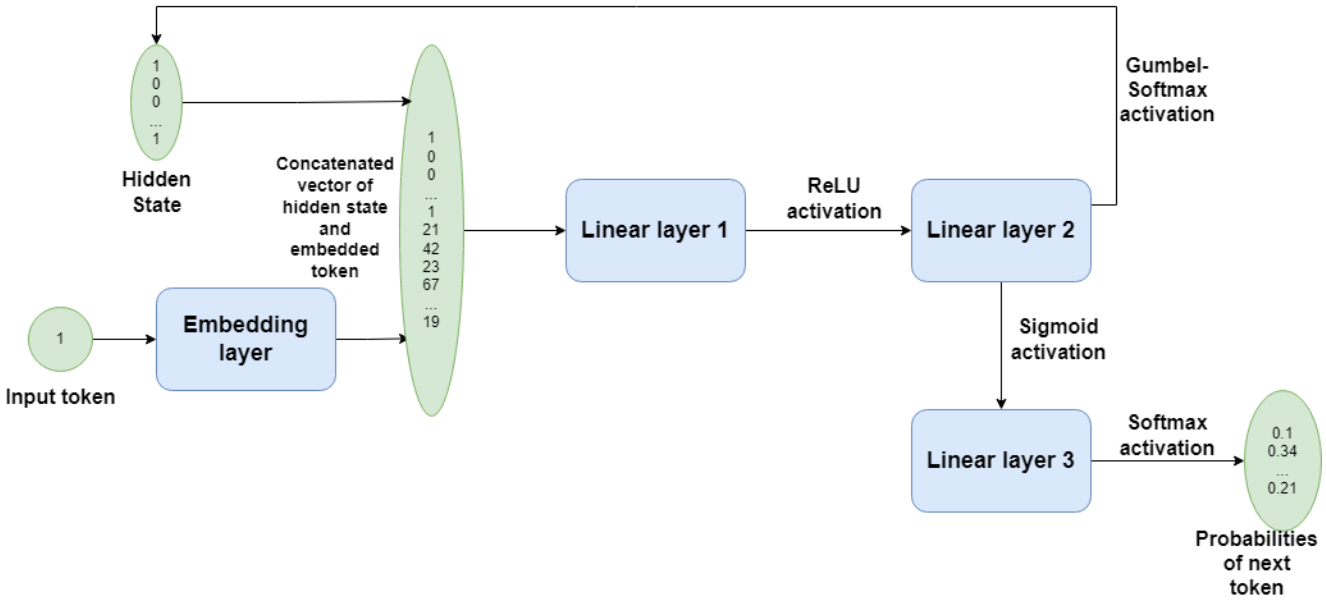
4.1 Implementation of Recurrent Neural Network

Let us take an event log from the example before $\boldsymbol{L} = [< a, b, c, d, e >^{14}, \ < a, b, b, c >^{12}]$. In order to be able to feed each of the traces in our RNN, they have to be padded. The padding adds a special symbol "_" to each of the traces in order for them to be of equal size. We also want to make each trace longer by one token, to get the desired number of states at the final output of RNN. So, after padding, the log will look the following way: $\boldsymbol{L} = [< a, b, c, d, e, \_ >^{14}, \ < a, b, b, c, \_, \_ >^{12}]$. Each trace in the RNN will be then the iterated $\boldsymbol{(n-1)}$ times, where $\boldsymbol{n}$ is the length of padded trace in event log (given that they are all now of equal size). Events in traces are also encoded, each being mapped to a positive integer when fed into RNN.

We now proceed to describing the architecture of RNN, that yields us the outputs required for building the transition system. At each step RNN should accept a single token from a trace as an input and then produce the vector of probabilities for each of the tokens present in the event log as the output. Since event log contains certain patterns and the upcoming events in a trace are closely related to the preceding event, model should preserve information about the previous event and pass it to the next step. This is why we are using a Recurrent Neural Network. The hidden state will be designed in such a way, that it will represent the states of our future transition system.

At the beginning of each iteration a single token from a trace is fed into RNN. Then, it passes through the embedding layer, thus being encoded into a vector of values. The embedded token is concatenated with the hidden state from the previous step, if it is the first iteration for a trace, then hidden state is initialized with a vector of zeros, and then passed to the first linear layer with an activation function ReLU. The number of neurons in this layer is free to choose, not depending on any other things.

The second linear layer produces two outputs, one for the next hidden state and the other fed into the third linear layer. Its output size corresponds to the size of hidden state. Two activation functions are applied in the second linear layer, one being Gumbel-Softmax, which produces a binary vector, which is passed as a hidden state on the next iteration, and the other one is a sigmoid function passed to a third linear layer. By adding a third linear layer we are not limiting the size of our hidden state, since the size of a vector with event probabilities should always be of size of unique events in given event log. After the third layer we apply a softmax function, to return a vector of probabilities. Below is the schematic picture of built RNN.



The next step is to define the loss function, which would allow for efficient training. As we approach the task as classification, we may use Categorical Cross-entropy loss. At the end of each trace we calculate the error function and obtain a vector of scalar losses for each event. The calculation of Categorical Cross-entropy loss have been defined earlier. However, one modification have been introduced. As all of the initial traces in event log are padded, they contain meaningless events, which were added to fit the input criteria and get the desired number of states for building Transition System. It would be harmful to take padded events into account, and we

definitely don't want our neural network to learn to predict them either. Therefore, we zero out the error functions in the places where our model predicts padded events.

Training of RNN is done using stochastic gradient descent and afterwards all of the traces in the event log are fed into trained model in order to obtain the set of states.

4.2 Building the Transition System

After obtaining set of states for each trace in the event log, a transition system have to be built. It is going to be built according to the definition in one of the previous section. So, first off, convert the states set from binary vectors into a Natural number. Converting vectors of states we get a collection of sequences of states corresponding to traces in the event log. Going iteratively through sequences of states we will obtain a set of unique states ***S*** in our transition system TS, and in the same manner derive the set of events ***A.***

Finally, we start building a set of transitions but beforehand each sequence of states must be updated with an initial state **s0**. At each step, we will take a trace and a corresponding sequence of states to it and extract a set by transitions (a transition $s_i \times a_i \times s_{i+1}$ will be constructed by taking the **i** and the **(i+1)** state from sequence of states and the i event from trace). The last state from each sequence will be added to the set of final states S^{end}.

5 Results

The following approach with the Recurrent Neural Network allows us to build Transition Systems, which are perfectly fit. We may replay any trace in our event log due to the design of approach. One of the most important parameters in the RNN is

the size of the hidden state. Since it has binary representation an increase of hidden

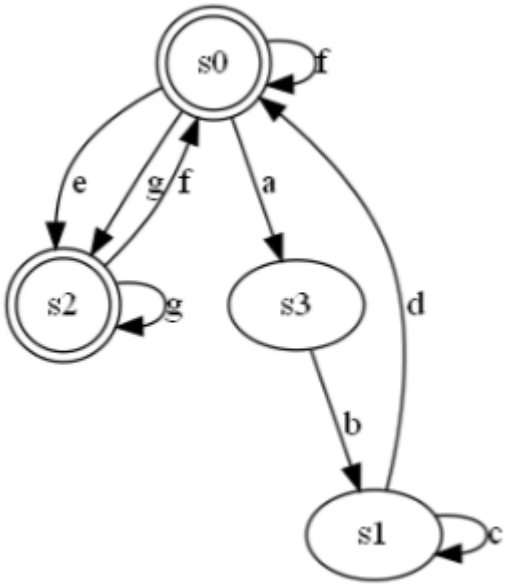


Figure 5.1 Transition System built by
RNN with hidden size 2.

state vector size by 1 potentially increases the maximum possible number of unique states by the power of two. Overall, as the size of hidden state increases, the models precision also tends to increase. By introducing more states Recurrent Neural Network starts to capture more and more patterns and allows for much less deviation. This can be in the pictures above and below. Figure 5.1 shows the Transition System built by RNN with hidden size 2, the maximum number of unique states is four in this case so our Transition System is very limited. As a result, we obtain a simple TS, capable of reproducing the log, yet allowing for too much redundant behavior. Figures 5.2 where hidden state is of size 10 represent a much better TS, being good in both simplicity and not overfitting. It still allows for some deviations but they are at acceptable level. Increasing the size of hidden state up to 80 shows us an overffited TS. It is only possible to replay the traces present in the event log, leading to a precision of 1 but a low simplicity.

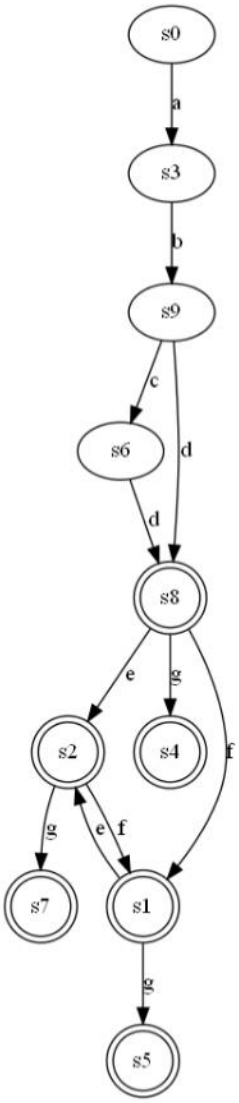


Figure 5.2 Transition System built by RNN with hidden size 10

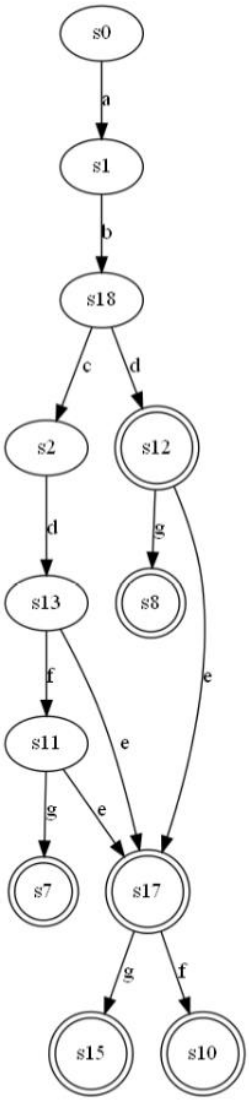


Figure 5.3 Transition System built by RNN with hidden size 80

6 Conclusion

Overall, the RNN shown a good job at detecting the patterns and built a relatively good Transition Systems. One of the main problem is still a long time to train and the requirement to fit the hyperparameters, which would yield a good fit for given event log. For larger event logs, this is of a big problem, as it would be impossible for a human eye to determine whether it is overfit of underfit. In the future, we may

produce some tests and try to detect dependency between the given event log and the required size of a hidden state to get a simple, yet precise process model as an output.

7 References

[1] A Unified Gradient-Descent/Clustering Architecture for Finite State Machine Induction, Sreerupa Das and Michael C. Mozer.

[2] Discovering Models of Software Processes from Event-Based Data, JONATHAN E. COOK and ALEXANDER L. WOLF

[3] Neural Approach to the Discovery Problem in Process Mining, Timofey Shunin(B), Natalia Zubkova, and Sergey Shershakov

[4] CS 224D: Deep Learning for NLP, Lecture Notes: Part III, Author: Rohit Mundra, Richard Socher

[5] Categorical Reparameterization with Gumbel-Softmax, Eric Jang, Shixiang Gu, Ben Poole

[6] Code of the project