University of Bergen
Department of Informatics

Building a finite state automaton for physical processes using queries and counterexamples on long short-term memory models

Author: Eivind Anton Sætre Skarstein

Supervisor: Rodica Mihai

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

Most neural networks (NN) are commonly used as black-box functions. A network takes an input and produces an output, without the user knowing what rules and system dynamics have produced the specific output. In some situations, such as safety-critical applications, having the capability of understanding and validating models before applying them can be crucial. In this regard, some approaches for representing NN in more understandable ways, attempt to accurately extract symbolic knowledge from the networks using interpretable and simple systems consisting of a finite set of states and transitions known as deterministic finite-state automata (DFA).

In this thesis, we have considered a rule extraction approach developed by Weiss et al. that employs the exact learning method L* to extract DFA from recurrent neural networks (RNNs) trained on classifying symbolic data sequences. Our aim has been to study the practicality of applying their rule extraction approach on more complex data based on physical processes consisting of continuous values. Specifically, we experimented with datasets of varying complexities, considering both the inherent complexity of the dataset itself and complexities introduced from different discretization intervals used to represent the continuous data values.

Datasets incorporated in this thesis encompass sine wave prediction datasets, sequence value prediction datasets, and a safety-critical well-drilling pressure scenario generated through the use of the well-drilling simulator OpenLab and the sparse identification of nonlinear dynamical systems (SINDy) algorithm.

We observe that the rule extraction algorithm is able to extract simple and small DFA representations of LSTM models. On the considered datasets, extracted DFA generally demonstrates worse performance than the LSTM models used for extraction. Overall, for both increasing problem complexity and more discretization intervals, the performance of the extracted DFA decreases. However, DFA extracted from datasets discretized using few intervals yields more impressive results, and the algorithm can in some cases extract DFA that outperforms their respective LSTM models.

Acknowledgements

I would like to thank my supervisor Rodica Mihai for her support and guidance throughout this thesis. Your insightful feedback and expertise have been invaluable.

I am very thankful to NORCE, OpenLab for granting me access to their drilling simulator through their academic license. Their simulator has been essential for allowing me to conduct experiments using both interesting and high-quality data.

Lastly, I would extend my sincere gratitude to my friends and family for supporting me throughout this endeavor. I am especially grateful for my patient and loving girlfriend, who has been an endless source of inspiration and motivation.

Eivind Anton Sætre Skarstein
Thursday 1st June, 2023

Contents

List of Abbreviations **1**

1 Introduction **2**

1.1 Problem Statements and Objectives 4

1.2 Research Questions 4

1.3 Thesis Outline 4

2 Background **6**

2.1 Neural Networks for Learning Sequence Data 6

2.1.1 Artificial Neural Networks 6

2.1.2 Recurrent Neural Networks 7

2.1.3 Long Short-Term Memory 10

2.2 Deterministic Finite-state Automata 11

2.3 Rule Extraction 12

2.3.1 Rule Extraction from RNN 13

2.3.2 L* Algorithm 14

2.3.3 Rule Extraction using L* Algorithm 15

2.4 OpenLab 16

2.5 Sparse Identification of Nonlinear Dynamical Systems 18

2.6 F1 Score 20

2.7 Key Python Libraries 21

2.7.1 PyTorch 21

2.7.2 PySINDy 22

3 Methodology and Approach **23**

3.1 Dataset Generation 23

3.1.1 Discretization 24

3.1.2 Weights 25

3.1.3 Sine Wave Dataset 25

3.1.4 Sequence Value Dataset . . . 27
3.2 Well-Drilling Dataset . . . 28
3.2.1 Problem Description . . . 29
3.2.2 OpenLab Data Generation . . . 30
3.2.3 Synthetic Data Generation using PySINDy . . . 32
3.2.4 Data Generation and Sequence Labelling . . . 33
3.3 Workflow Pipeline . . . 34
3.3.1 Models . . . 35
3.3.2 Training Script . . . 35
3.3.3 DFA extraction . . . 36
3.3.4 Evaluation Script . . . 38
3.4 Summary . . . 38

4 Experiments and Results **40**
4.1 Experimental Settings . . . 40
4.2 Evaluation Metrics . . . 41
4.3 Extraction Parameters . . . 41
4.3.1 Extraction Time . . . 41
4.3.2 Initial Refinement Depth . . . 42
4.3.3 Starting Examples . . . 42
4.3.4 Effect of Extraction Parameters . . . 42
4.4 Method Limitations . . . 45
4.5 Test Case Results for Self Generated Datasets . . . 47
4.5.1 Sine Wave Dataset . . . 47
4.5.2 Sequence Value Dataset . . . 52
4.6 Test Case Results for the Well-Drilling Dataset . . . 56
4.6.1 Sparse Identification of Nonlinear Dynamical Systems . . . 56
4.6.2 Experiment Configuration . . . 58
4.7 Size and Inference Time Comparisons . . . 60

5 Discussion and Analysis of Results **62**
5.1 Sparse Identification of Nonlinear Dynamical Systems . . . 62
5.2 Datasets . . . 63
5.3 Evaluation Metric . . . 65
5.4 Extracted DFA Discussion . . . 65
5.5 Future Work . . . 66

6 Conclusion **68**

Bibliography **70**

A LSTM Model Code **75**

List of Figures

2.1 Unfolded RNN. 9
2.2 Illustration of gated cells in an LSTM cell. 11
2.3 Finite automata recognizing the word "this" over an alphabet y. All sequences that do not form the word "this" are rejected. 12
2.4 OpenLab web environment. Screenshot taken from OpenLab (2023a) . . 17
2.5 SINDy algorithm applied on data collected from the Lorentz equations. Figure taken from (Brunton et al., 2016). 19

3.1 Finite set of values discretized from continuous values ranging from -1 to 1, and the corresponding alphabet symbol for each value. 24
3.2 Original sine wave and discretized datasets 26
3.3 The labeling and sequence value of each sequence step for 2000 sequences using an alphabet of size 10. Not all sequences have the same length. . . 27
3.4 The labeling and sequence value of each sequence step for 2000 sequences using an alphabet of size 10. After 10 steps all sequences increase in value. 28
3.5 Possible geological pressure constraints to a depth of 5000m. s.g. is the specific gravity, tvd is true vertical depth. Screenshot taken from OpenLab (2023a). 29
3.6 Bottom hole pressure versus inputs of flow rate and rpm. 31
3.7 SINDy generated sequences based on simulated data from OpenLab. Dots show sampling points. 33
3.8 SINDy generated sequences based on simulated data from OpenLab. Dots show sampling points. 34
3.9 Workflow with external resources. 38

4.1 Extracted DFA using an alphabet of size 3. 44
4.2 Correct DFA extracted from 4.5.1. Illustrates the ground truth of the sine dataset using an alphabet size of 3. 48

4.3 Correct DFA extracted from 4.5.1. Only sequences that form a part of a sine wave are accepted. This illustrates the ground truth of the sine dataset using an alphabet size of 5. 50
4.4 Incorrect DFA extracted from 4.5.1. 51
4.5 2000 sequences using an alphabet of size 3. 52
4.6 2000 sequence using an alphabet of size 25. 52
4.7 Sequence dataset using an alphabet of size 25. After 10 steps the "value" of each sequence is affected by external influence and increased by 5. . . . 53
4.8 Extracted DFA using using alphabet size 10 on the action value dataset. 13 nodes. 54
4.9 SINDy generated BHP sequence compared to OpenLab simulated BHP sequence. 57
4.10 Full SINDy generated BHP sequences based on simulated data from OpenLab. k=1. 58
4.11 Sampled SINDy generated BHP sequences based on simulated data from OpenLab. k=1 59
4.12 Percentage prediction distribution for each classification label. 60

List of Tables

4.1 F1 scores of DFA and LSTM model for different extraction times. 43
4.2 F1 scores of DFA and LSTM model for different refinement depths. . . . 43
4.3 F1 scores of DFA and LSTM model for different length starting examples. 43
4.4 Average number of nodes for extracted DFA and resulting F1 score on the test dataset. 46
4.5 Performance of models and extracted DFA for the sine dataset. 49
4.6 Performance of models and extracted DFA, no external influence. 54
4.7 Performance of models and extracted DFA, with external influence. . . . 55
4.8 Performance of models and extracted DFA, on the well-drilling dataset. . 59
4.9 Table comparing F1 score, size, and inference time between extracted DFA and LSTM models. 61

Listings

A.1 LSTM model code 75

List of Abbreviations

ANN Artificial Neural Network.

API Application Programming Interface.

BHP Bottom Hole Pressure.

CPU Central Processing Unit.

DFA Deterministic Finite-state Automata.

FA Finite-state Automata.

GPU Graphics Processing Unit.

LSTM Long Short-Term Memory.

MLP Multilayer Perceptron.

NN Neural Network.

RNN Recurrent Neural Network.

ROP Rate of Penetration.

RPM Revolutions per Minute.

SINDy Sparse Identification of Nonlinear Dynamical Systems.

SVM Support Vector Machine.

WOB Weight on Bit.

Chapter 1

Introduction

Neural networks (NN) are powerful tools for processing large quantities of data with limited human assistance. These networks are inspired by the structure and function of the human brain, emulating the ability to learn, adapt and make decisions using interconnected neurons (Rosenblatt, 1958). Because of this, NNs are capable of learning sophisticated models able to, for example, classify, predict, and cluster large sets of complex data. As a result, NNs have been applied with considerable success in challenging tasks such as image recognition, speech processing, and natural language processing (Abiodun et al., 2018).

However, despite many advancements within the field of NN, most NN models are treated as black-box functions, with the user unable to know anything about the rules that govern the models decision-making. The black-box nature of NNs is mainly due to the fact that the NN architecture is designed to store learned knowledge in its weights, and as a result makes it difficult to near-impossible to inspect, analyze and validate the information (Omlin and Giles, 2000). In certain domains it is necessary to validate a model before using it, requiring a certain degree of understanding of the model. Some industrial safety-critical applications where model validation is crucial are, for example, self-driving cars, malware detection, and aircraft collision avoidance systems (Xiao et al., 2021).

Due to this, a significant amount of work has been devoted to representing NN architectures in more understandable ways. Particularly, several approaches have been developed to extract symbolic knowledge efficiently and accurately from trained recurrent neural networks (RNNs)(Jacobsson (2005); Wang et al. (2018a)). RNNs are a type of NN that excels in processing sequential data where the order of the data points is important. These networks are able to capture the temporal dependencies between data points in the sequence, allowing them to learn sequence dynamics, and model complex patterns in the data (Schmidt, 2019).

As a consequence of the stateful nature of RNNs, prevalent methods for rule extraction aim to derive symbolic knowledge from trained RNNs by constructing an automaton that emulates the model. Automata allows for suitable interpretations of an RNNs decision logic, by defining the explicit states and transition rules of the model (Zhang et al., 2021b). The simplest kind of automaton, and a common choice for representing extracted rules, is a Deterministic Finite-state Automata (DFA) (Wang et al., 2018b). A DFA is a system of finite states consisting of clearly defined transition functions, used to recognize patterns in sequences consisting of a finite set of symbols (Carroll and Long, 1989).

Weiss et al. (2018) have developed an approach for conducting rule extraction on RNNs trained on classifying symbolic data sequences based on a specific set of rules known as regular languages. Their approach uses the exact learning method L* developed by Angluin (1987) to represent the underlying system dynamics of a given trained RNN as a DFA. Weiss et al. demonstrate that their method in many cases, produces faster and more accurate DFA compared to other extraction methods when evaluated on multiple benchmark datasets, and in some cases, is able to extract DFA that outperforms the RNN on test data.

However, many real-world applications collect and analyze data composed of sequences of continuous values, such as weather forecasting, stock price forecasting, and housing price predictions, in contrast to sequences of symbolic data. In this thesis, the goal is to study how well extraction methods meant for regular language data can be implemented for models trained on datasets aimed at representing real-world processes, with a specific focus on the rule extraction approach presented by Weiss et al. Being able to apply extraction methods to models trained on such data will enable a deeper comprehension of these models. Additionally, for our study, we utilize the Long Short-Term Memory (LSTM) RNN architecture. Evaluations are conducted from an assortment of datasets of varying complexity. Particularly in this thesis, we examine sequence data from physical processes and will employ their algorithm on self-generated sequential datasets and datasets generated through a combination of the virtual drilling simulator OpenLab (Ewald et al., 2018; Saadallah et al., 2019) and the sparse identification of non-linear dynamics algorithm (Brunton et al., 2016).

1.1 Problem Statements and Objectives

The objective of this thesis is to expand upon previous work on rule extraction from trained RNNs done by Weiss et al. (2018), by applying and evaluating their approach on sequence data from models based on physical processes. Mainly, we experiment with extracting multiple DFA from LSTM networks trained on datasets of different-sized alphabets, where we explore various extraction parameters such as refinement depth, starting example length, and extraction time. We evaluate the quality of extracted DFA compared to the trained LSTM network primarily by analyzing their performance on training data, unseen test data, and long test data consisting of sequences longer than the LSTM was trained on.

1.2 Research Questions

We aim to explore the following research questions:

RQ1. How well will the rule extraction algorithm designed for use on regular languages transition to more complex data describing physical processes?

RQ2. How can the rule extraction algorithm be applied to models trained on sequences of continuous data values?

RQ3. In what ways will hyperparameter selection affect the accuracies and extraction time of the method for our datasets?

1.3 Thesis Outline

Chapter 2: Background: We discuss the prerequisite theory for understanding the domains and concepts in this thesis, such as recurrent neural networks, deterministic finite-state automata, and exact learning.

Chapter 3: Methodology: This chapter covers the technical work of preparing the experiments, including data generation, model implementations, and the training pipeline.

Chapter 4: Experiments and results: We present multiple test cases and their results. DFA are extracted from LSTM models trained on sine wave prediction datasets, sequence value datasets, and synthetic well-drilling datasets generated from a combination of OpenLab and the sparse identification of nonlinear dynamical systems algorithm.

Chapter 5: Discussion and analysis: We summarize the work and discuss results based on the previous chapter, and present avenues for interesting future research and exploration.

Chapter 6: Conclusion

Chapter 2

Background

This chapter gives an overview of relevant concepts and methods central to understanding the rule extraction approach explored in this thesis. Section 2.1 details neural network algorithms for processing sequence data. Section 2.2 introduces the concept of deterministic finite-state automata. Section 2.3 presents rule extraction and outlines the main approach employed in this thesis. In section 2.4 we present the drilling simulator OpenLab. Section 2.5 describes the method of sparse identification of nonlinear dynamical systems used for data generation. Section 2.6 presents the F1-score evaluation metric. Lastly, section 2.7 introduces PyTorch and PySINDy, two important Python libraries used in this thesis.

2.1 Neural Networks for Learning Sequence Data

This section presents popular algorithms used to learn from sequence data. Equations presented in this section follow the notation by Zhang et al. (2021a). Additionally, Zhang et al. (2021a) is used as inspiration for the illustrations presented in this section.

2.1.1 Artificial Neural Networks

Essentially, an artificial neural network (ANN) is a network consisting of connected computing units commonly referred to as neurons(Gamboa, 2017). The neurons carry out computations and transmit the resulting output values through their outgoing connections to other neurons. Each connection has a weight that dictates how strongly the two neurons are connected. The calculations of the neurons start by summing together the inputs from all incoming connections with the addition of a bias term to the sum before passing the value to an activation function. An activation function is used to decide whether the computation should be passed on or not to the next neurons (Zhang et al., 2021a). A wide variety of activation functions have been developed, some of the most common are the sigmoid function and the rectified linear unit (ReLU) function (Rasamoelina et al., 2020).

A network is organized in layers of neurons. The input layer receives the input, the output layer produces the network's output, and all layers in between that are not directly exposed to the environment are known as the hidden layers. Hidden layers can be stacked to form a deep network, and each layer can have any number of neurons, known as hidden neurons. A common and simple ANN architecture is the multilayer perceptron (MLP) (Rosenblatt, 1958). MLPs are characterized by transmitting the outputs within the network in one direction, containing no loops, and in principle, there is no way for the output of the units to affect themselves. This is known as a feedforward network (Popescu et al., 2009). The equation for computing the layer outputs and the final output of a MLP can be written as:

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}_x + \mathbf{b}_h), \tag{2.1}$$

$$\mathbf{O} = \mathbf{H}\mathbf{W}_o + \mathbf{b}_o \tag{2.2}$$

Here $\mathbf{H}$ and $\mathbf{O}$ are the output of the hidden layer and the output layer respectively, $\mathbf{X}$ is the input matrix, $\mathbf{W}$ is the weight matrices for the input matrix for each layer, $\mathbf{b}$ is the bias term, and σ is an activation function. For deeper networks, we can compute the output of new layers by using the output $\mathbf{H}$ of the previous layer as input $\mathbf{X}$ for the next layer.

2.1.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are an architecture of neural networks primarily used for processing sequential data. Sequential data can span a diverse range of applications such as handwritten characters, genetic sequences, text, and numerical time series generated in industrial settings (Schmidt, 2019). Through the use of recurrent connections, or cycles, that transmit information back into the network's previous layers, RNNs expand upon the capabilities of feedforward networks, such as the MLP, by considering previous inputs in addition to the current input. Due to this, RNNs can effectively capture the dynamics of sequences, including remote influences within the sequence. One way to define an RNN is in a similar manner as an MLP in equation 2.1:

$$\mathbf{H}_t = \sigma(\mathbf{X}_t\mathbf{W}_x + \mathbf{H}_{t-1}\mathbf{W}_h + \mathbf{b}_h) \tag{2.3}$$

$$\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_o + \mathbf{b}_o \tag{2.4}$$

Here $\mathbf{H}_t$ is the hidden layer output at time step t, $\mathbf{H}_{t-1}$ is the output from the previous layer, and the term $\mathbf{X}_t$ represents the input at each time step. The addition of $\mathbf{H}_{t-1}$ makes the computation recurrent.

Training

A key principle of NN architectures is the ability to learn from data. To achieve this, NNs update the weights between neurons iteratively through a process called model training. For this thesis, we employ labeled datasets, where each input has a corresponding output[1]. Training encompasses two parts, forwardpropagation, and backpropagation.

During forwardpropagation, the output of a NN for a given input is computed. While backpropagation adjusts the weights of the network to minimize the difference between the computed output and the wanted output in the dataset. The equation for computing outputs in a feedforward neural network and an RNN is given in equations 2.1 and 2.3 respectively. During backpropagation, the loss gradient is used to update the model parameters in the neural network. This is achieved by first computing the gradient for the loss function L with respect to a given parameter ϕ as shown in equation 2.5, with $\mathbf{H}$ being an output from a layer. Afterwards, the chain rule is used to extend the equation and compute the loss gradients for all parameters throughout the network.

$$\frac{\partial L}{\partial \phi} = \frac{\partial L}{\partial \mathbf{H}} \cdot \frac{\partial \mathbf{H}}{\partial \phi} \tag{2.5}$$

RNNs can be trained using forwardpropagation and backpropagation in a similar manner to feedforward networks. By unfolding an RNN, it can be visualized as a feedforward network with shared parameters between layers. This approach allows for the use of the same training process as feedforward networks, however with some adaptations. Notably, each parameter appears once for each timestep when computing variables, and depending

[1]Training in this fashion is known as supervised learning (Zhang et al., 2021a)

on the RNN architecture, an output may be computed for each timestep contributing to the total loss. In figure 2.1 an unrolled RNN is depicted. We compute the gradient of

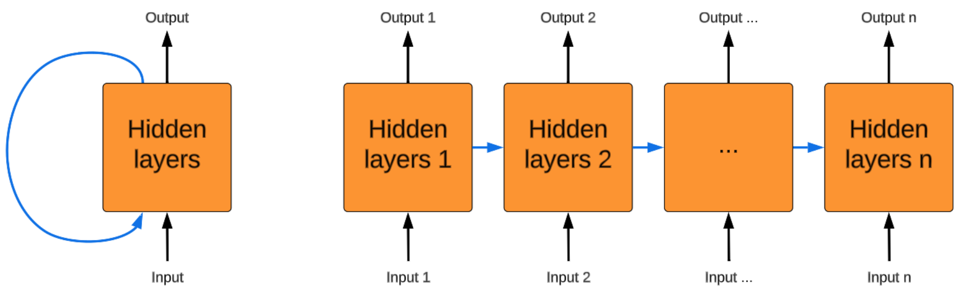


Figure 2.1: Unfolded RNN.

the loss L for an RNN with regards to a parameter ϕ as:

$$\frac{\partial L}{\partial \phi} = \sum_{t-1}^{T} \left(\frac{\partial L}{\partial \mathbf{H}_t} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi} \right) \tag{2.6}$$

As a result of $\mathbf{H}_t$ being dependent on the output $\mathbf{H}$ from previous time steps we can write the term $\frac{\partial L}{\partial \mathbf{H}_t}$ as:

$$\frac{\partial L}{\partial \mathbf{H}_t} = \frac{\partial L}{\partial \mathbf{H}_{t+1}} \cdot \frac{\partial \mathbf{H}_{t+1}}{\partial \mathbf{H}_t} \tag{2.7}$$

In theory, we have unfolded the RNN to a feedforward NN and performed backpropagation. This adaptation of backpropagation for RNNs is known as backpropagation through time.

Vanishing Gradients

RNNs, like most neural networks, can be susceptible to experiencing vanishing or exploding gradients (Schmidt, 2019). Recall equation 2.7 for computing gradients in an RNN, the term $\frac{\partial \mathbf{H}_{t+1}}{\partial \mathbf{H}_t}$ introduces matrix multiplication over the sequence and can significantly impact cases with long sequences depending on the size of its values. If the values are large (>1), it can lead to an exploding gradient, where each weight is modified excessively, rendering model learning impossible. Conversely, if the values are small (<1), the gradient decreases with each time step, resulting in a vanishing gradient that eliminates the influence of earlier time steps.

The problem of vanishing gradients poses a significant challenge for RNNs and restricts their effectiveness in handling long sequences. To overcome this issue, modern RNN architectures have been developed to improve performance on long sequences. The Long Short-Term Memory algorithm is one of the most popular RNN architectures for this purpose.

2.1.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) is an RNN architecture designed to reduce the effect of vanishing gradients during the training process. LSTM networks are able to learn over longer sequences by incorporating more information into the network by using dedicated mechanisms for when to update and reset hidden states, known as gated cells (Zhang et al., 2021a). Input gates are used to determine how much of the input's value should be added to the cell. Forget gates decides what information should be removed from the cell. Lastly, the output gate controls how much of the cell will influence the output. The gated cells allow the network to learn what information is important and thus able to maintain long-term dependencies and avoid the vanishing gradient problem. The equations for the gates are given as:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i) \tag{2.8}$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f) \tag{2.9}$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o) \tag{2.10}$$

Where $\mathbf{W}$ and $\mathbf{b}$ are weights and biases respectively. $\mathbf{X}$ is the input and $\mathbf{H}_{t-1}$ is the hidden state computed in the previous time step.

New data for the network is computed using an input node $\widetilde{\mathbf{C}}$:

$$\widetilde{\mathbf{C}}_t = \mathbf{tanh}(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c) \tag{2.11}$$

Using all equations so far, we can design the memory cell:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \widetilde{\mathbf{C}}_t \tag{2.12}$$

Here $\odot$ is the symbol for point-wise multiplication. We see that the input gate $\mathbf{I}$ control how much new data we take in via $\widetilde{\mathbf{C}}$, and the forget gate $\mathbf{F}$ decides how much of the old cell state $\mathbf{C}_{t-1}$ to use. Lastly, the hidden state $\mathbf{H}_t$ is computed:

$$\mathbf{H}_t = \mathbf{O} \odot \mathbf{tanh}(\mathbf{C}_t) \tag{2.13}$$

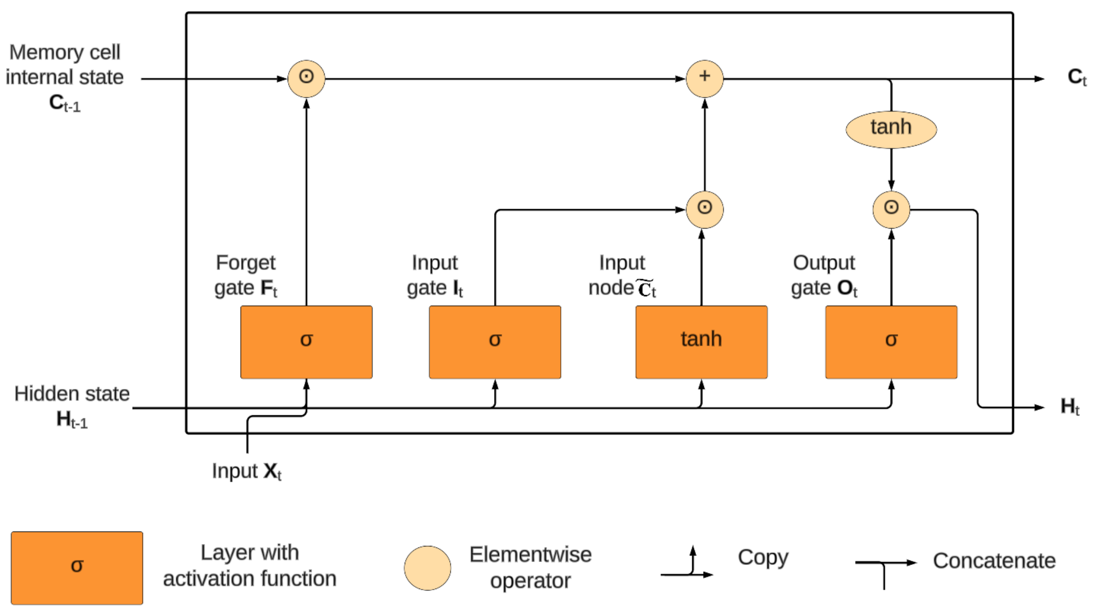


Figure 2.2: Illustration of gated cells in an LSTM cell.

The output gate **O** controls how much of the memory cell will influence the subsequent layers. Similar to hidden units, the weights for the gated cells are updated over the training process. This allows LSTM models to learn when to selectively incorporate or exclude distant data, and as a result, mitigate the problem of vanishing gradients. Figure 2.2 illustrates an LSTM cell.

2.2 Deterministic Finite-state Automata

Automata theory is the study of abstract computing devices and can be used to understand and formally represent computing systems and processes (Hopcroft et al., 2006). An automaton is a mathematical model consisting of states and transition functions. When given an input sequence, the automaton moves between states in response to each symbol in the sequence. There are different types of automata based on their complexity. The simplest automata consist of a finite set of states and are known as finite-state automata (FA).

Subsequently, the simplest class of FA is known as Deterministic Finite-state Automata (DFA) and is characterized by the property of using only a single unique transition per input. This is in contrast to the other common type of FA called Non-Deterministic Finite-state Automata, which allow for multiple transitions per input. DFAs are mathematical models that accept or reject words made from symbols taken from an alphabet

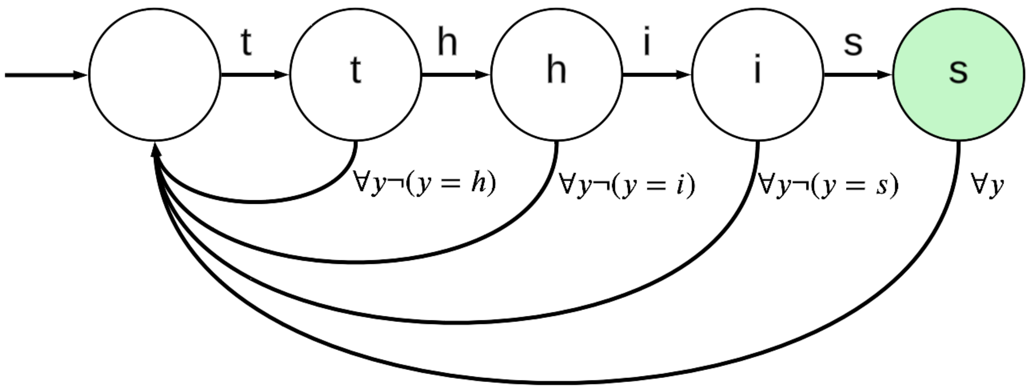


Figure 2.3: Finite automata recognizing the word "this" over an alphabet y. All sequences that do not form the word "this" are rejected.

(Carroll and Long, 1989). In this case, a word is a finite sequence of symbols from the alphabet. For example, "11001" is a word from a binary alphabet {0,1} (notice that even though we refer to the sequence as a word, it is not restricted to solely consist of letters).

A DFA is defined as a 5-tuple (Q, Σ, δ, q_0, F), where Q is a finite set of states, Σ is a finite set of input symbols, δ is a transition function that maps each state and input symbol to a new state, q_0 is the initial state $q_0 \in Q$, and F is a set of final states; $F \subset Q$ (Hopcroft et al., 2006). Figure 2.3 shows a small DFA accepting the word "this". All other inputs than those shown at each state result in the rejection of the word.

Regular Languages Regular languages are defined as languages that can be accepted by a DFA or expressed as regular expressions. Regular expressions are algebraic descriptions of the rules for a language and how a word is accepted. For example, the regular expression {1*0*} denotes a language that accepts all strings with no "1" to the right of a "0" over an alphabet $\Sigma = \{0, 1\}$ (Hopcroft et al., 2006).

2.3 Rule Extraction

NN models are inherently difficult to interpret and understand what governs its decision-making and are consequently used as black-box functions in many cases. Understanding such models is not only interesting from a scientific point of view but has important use cases in safety-critical domains where model validation is necessary prior to use in applications, such as for instance medical diagnosis and aircraft control (Wang et al., 2018b). In order to understand NN models, multiple approaches have been developed and experimented with. Here we focus on the approach known as rule extraction, where a DFA is extracted from a trained RNN model. Furthermore, in production, utilizing simplified models can be less expensive than running RNN models (Murdoch and Szlam, 2017), and require significantly less memory storage. This section covers the main approaches of rule extractions from RNNs, with a focus on the approach by Weiss et al., which utilizes the exact learning algorithm L*.

2.3.1 Rule Extraction from RNN

RNNs, like most NN, are notoriously difficult to interpret, analyze, and in general, understand their reasoning. The complexity of RNN with regards to interpretability stems from the fact that RNNs are designed to store learned knowledge within the weights of its network (Omlin and Giles, 2000). Compared to RNNs, DFA models consist of a fixed set of states and state transition rules, which can be expressed in a compact and easily interpretable form. Therefore, representing RNNs as DFA can be beneficial by simplifying the model and enhancing the understanding of the decision-making process while reducing the computational complexity of the RNN.

Rule extraction from RNNs is an old problem. Approaches for extracting DFAs from trained RNNs go as far back as to Minsky 1967, whom as part of his work discusses the concept of equivalence between finite-state machines and neural networks. Since then, multiple approaches have been introduced attempting to learn the internal logic of RNNs efficiently and accurately by representing it as automata. Jacobsson (2005) have done work in categorizing different methods for rule extraction from NNs. Particularly, in the case of RNNs with stateful model extractions, approaches mostly fit into two categories, compositional approaches, and pedagogical approaches (Zhang et al., 2021b).

Compositional Rule Extraction Approach

Compositional approaches are based on analyzing the hidden state space of a trained RNN and building up an automaton representing the behavior of the network using state and transition abstraction. The advantage of compositional approaches is that they allow for great scalability. However, the disadvantages are often unsatisfactory accuracies and the need for finding and selecting optimal parameters for the abstraction process (Zhang et al., 2021b). Roughly, approaches in this manner work by first clustering the hidden neurons of a trained RNN into states and then constructing a transition diagram based on the state connections. Lastly, the transition diagram is reduced to a minimal representation of state transitions (Wang et al., 2018b).

Pedagogical Rule Extraction Approach
In contrast, pedagogical approaches extract rules and build up automata aiming to represent the RNN as close as possible by directly using the model as a black-box function (Jacobsson, 2005). In this manner, pedagogical approaches make no assumptions about the internal configurations of the network and are thus flexible and applicable to all RNN architectures, and can even be applied to non-NN architectures, as for example, rule extraction from support vector machines as demonstrated by (Martens et al., 2008). Angulin's L* algorithm exemplifies a pedagogical approach to rule extraction, as it can employ any model capable of responding to membership and equivalence queries as a black-box function. L* is described in more detail in the next section.

2.3.2 L* Algorithm

The key principle behind the L* algorithm (Angluin, 1987) is to learn a DFA from a system by treating the DFA as a learner and the system as a teacher (Berg et al., 2005). The learner starts out with no knowledge about the system, and by asking queries to the teacher it builds up a hypothesis of a system representative DFA. In this case, the teacher is a minimally adequate teacher, which is able to answer two types of queries:

- **Membership query**: A word is either accepted or rejected by the teacher.
- **Equivalence query**: Determines whether a given hypothesis is equivalent to the system known by the teacher. If they disagree, a counterexample - a word where the hypothesis and system disagree – is offered.

In this manner, the learner gradually learns about the system. Typically, the learner builds up a hypothesized DFA by asking membership queries, and when confident about its hypothesis, asks an equivalence query to check if it is correct. If accepted by the equivalence query, the algorithm is finished, otherwise, a counterexample is returned where the hypothesized DFA and the system disagree. The cycle of membership queries and equivalence queries is repeated until the hypothesized DFA is accepted or until the process is stopped. The ultimate goal is to arrive at a DFA that accurately represents the system, or at least provides a good approximation. Furthermore, as a result of starting from an empty DFA and adding states and transitions in an incremental manner, the algorithm attempts to find the smallest possible DFA accurately representing the system. For the same reasons, the algorithm works in polynomial time.

2.3.3 Rule Extraction using L* Algorithm

The objective of this study is to expand upon the work of Weiss et al. (2018) by examining the application of their approach to more complex datasets based on real-world physical processes. Here, we present a summary of their methodology for rule extraction.

In section 2.3.2 we describe the exact learning algorithm L*, which involves a learner utilizing a minimally adequate teacher capable of answering membership and equivalence queries to learn a system known by the teacher. Weiss et al. (2018) present a method of combining the L* algorithm together with a binary-classification RNN trained on regular languages as a minimally adequate teacher, with the aim to learn a DFA that, as close as possible, accepts the same language as the RNN. A binary-classification RNN, often referred to as an RNN-acceptor, is able to label a sequence of data either true or false. Their evaluation of the method across multiple datasets demonstrates good performance in terms of both DFA extraction accuracy and extraction time, especially for datasets with small alphabets.

Combining the L* algorithm with RNNs presents certain challenges. Mainly, directly utilizing the RNN as the teacher answering the equivalence queries likely results in an intractable problem, where suboptimal techniques can lead to very time-consuming extractions. In their paper, Weiss et al. instead use an approximation and utilize a finite abstraction of the RNN as the teacher. In this case, the abstraction is an extracted DFA where every state is a partition of the state space of a given RNN with regard to a partition function p. Each transition and classification of the abstraction is defined by one sample from each partition. The abstraction and the extracted DFA are in this manner two hypotheses of the ground truth RNN. Prior to achieving equivalence with the RNN, equivalence must be achieved between the hypotheses. In the event of any contradictions between the hypotheses, the RNN is utilized to determine the correct classification and obtains either a counterexample to the DFA or a refinement of the abstraction.

Subsequently, their approach to answering the minimal adequate teacher queries can be described in the following manner:

- For answering a membership query, a given word is either accepted or rejected by the RNN-acceptor.
- For answering equivalence queries a finite abstraction from the RNN is used as a comparison with the DFA obtained from the L* algorithm. If the DFA and the

abstraction are not equivalent, the RNN is used as ground truth to find the true classification resulting in either refining the abstraction or returning a counter-example to the DFA. This continues until the DFA and the abstraction are equivalent.

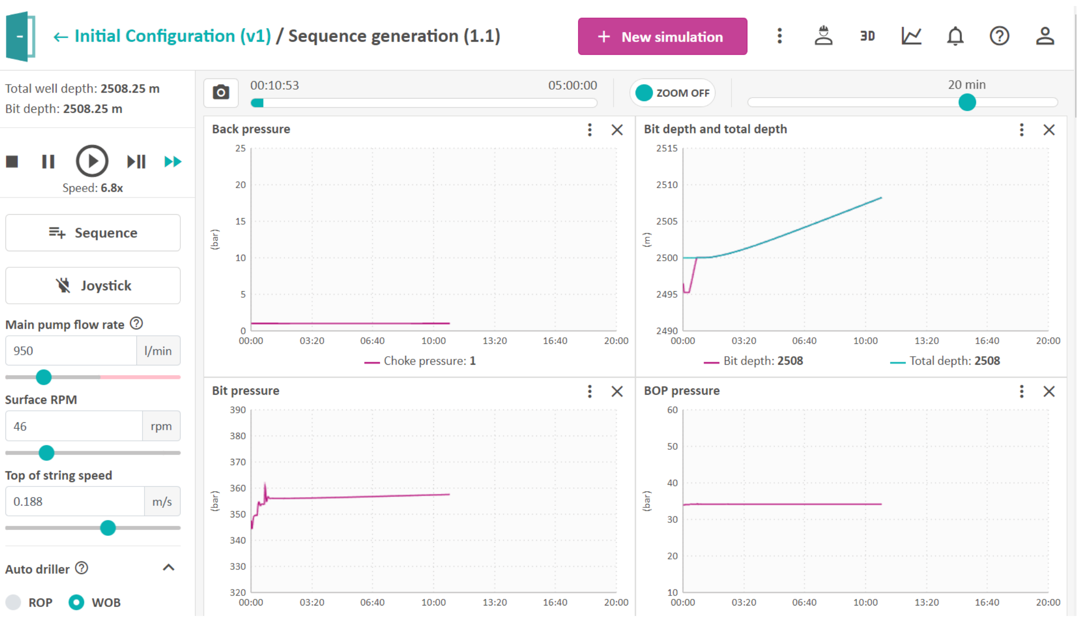
As a result, their approach will never return any incorrect counterexamples or force any unnecessary abstraction refinements. Their approach also supports early stopping and size limits, which is advantageous for cases with complex RNNs resulting in large DFA and time-consuming extractions.

Time-Consuming Abstraction Refinement

It is important to note that the size of the abstraction can influence the total extraction time of the rule extraction algorithm. The partitioning function used to define the abstraction is refined by separating the abstraction states using a Support Vector Machine (SVM) classifier. For each refinement, the SVM classifier is fitted on the state space of the RNN, and a new partition for separating a new state is defined. This increases the number of abstraction states by one. In this approach, an observation table is updated using membership queries on all the shortest sequences from the initial state to all states in the abstraction, with each sequence being extended by each symbol in the alphabet. The extended sequences are searched through as a breadth-first search. If there are inconsistencies between the classification made by the abstraction and the correct class of the extended sequence, a refinement is performed. Only when there are no inconsistencies is the abstraction and the hypothesized DFA presented to the teacher as an equivalence query. If accepted, the DFA is returned, if not a counter-example is returned and refinement of the automaton continues. However, the current hypothesized DFA is saved and returned if the extraction time exceeds a predefined time limit before finding a representative DFA. For these reasons, large abstractions and large alphabets mean more single-symbol extended sequences that must be verified. Possibly resulting in a very time-consuming abstraction exploration and refinement process.

2.4 OpenLab

For the construction of our well-drilling dataset, we utilize the state-of-the-art drilling simulator OpenLab, developed by the Norwegian Research Centre (NORCE) in collaboration with the University of Stavanger (OpenLab, 2023b). The OpenLab simulator has


← Initial Configuration (v1) / Sequence generation (1.1)

Total well depth: **2508.25 m**
Bit depth: **2508.25 m**

Speed: 6.8x

Main pump flow rate: 950 l/min

Surface RPM: 46 rpm

Top of string speed: 0.188 m/s

Auto driller: ROP [✓ WOB]

00:10:53 05:00:00 ZOOM OFF 20 min

Back pressure — Choke pressure: **1**

Bit depth and total depth — Bit depth: **2508** — Total depth: **2508**

Figure 2.4: OpenLab web environment. Screenshot taken from OpenLab (2023a)

been publicly available for students and researchers since 2018, and offers high-fidelity drilling simulations with a variety of options for overseeing and controlling the simulations. This include a user-friendly web application and a web Application Programming Interface (API) available in multiple programming languages (Saadallah et al., 2019). Figure 2.4 displays an active simulation in the OpenLab web environment.

The drilling process is simulated using several fully integrated numerical models. These models include a transient multi-phase flow model, a transient cuttings transport model, a torque and drag model, and a heat transfer model (Ewald et al., 2018). As such, the complex and realistic drilling simulations combined with the controllable and user-friendly environment make OpenLab a well-suited source for generating training and test datasets representing physical processes. OpenLab provides multiple license options with a varying degree of features. Their free version is available for everyone with an account and gives access to limited simulation features. In this thesis, we have received OpenLab's academic license which supports longer and multiple simultaneous simulations.

Although OpenLab supports multiple access options for the simulated drilling data, in this thesis we have exclusively employed the web API in Python for simulations and generating datasets. In general, conducting simulations in OpenLab start out with defining an initial drilling configuration. The configuration allows for setting a vast amount of drilling parameters. Some of these parameters include well path design, drillstring components, rig type, formation strengths, start depth, drilling fluid, and geology design.

Subsequently, the process of running OpenLab simulations entails providing setpoints to the simulator. Setpoints are control variables for the simulation and encompass drilling parameters, such as flow rate, Revolutions per Minute (RPM), Weight on Bit (WOB), and surface velocity.

2.5 Sparse Identification of Nonlinear Dynamical Systems

Additionally, for the generation of our well-drilling dataset, we leverage the Sparse Identification of Nonlinear Dynamical Systems (SINDy) algorithm on simulation data obtained from OpenLab. SINDy is a method for extracting the governing equations of nonlinear dynamical systems by employing sparsity-promoting techniques (Brunton et al., 2016). Consider a dynamical system on the form:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)). \tag{2.14}$$

Here $\mathbf{x}$(t) is the state of the system at time t, and the function $\mathbf{f}(\mathbf{x}(\mathrm{t}))$ represents the changes of the system over time. The function $\mathbf{f}$ is assumed to consist of only a few terms, making it sparse when considering all possible functions. In turn, this assumption allows for applying sparse regression algorithms to find sparse solutions describing the system. $\mathbf{f}$ is determined using temporal data from $\mathbf{x}$(t) and its derivative with respect to time $\dot{\mathbf{x}}$(t), either measured or numerically approximated from $\mathbf{x}$(t). The data is sampled at times t_1, t_2 ... t_m, and can be described as the following matrices:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_m) \end{bmatrix} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{bmatrix} \tag{2.15}$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}^T(t_1) \\ \dot{\mathbf{x}}^T(t_2) \\ \vdots \\ \dot{\mathbf{x}}^T(t_m) \end{bmatrix} = \begin{bmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \cdots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \cdots & \dot{x}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \cdots & \dot{x}_n(t_m) \end{bmatrix} \tag{2.16}$$

We define a library $\Theta(\mathbf{X})$ of candidate functions that we assume the solution might consist of. The library could consist of a multitude of different terms, for example, constant, polynomial, and trigonometric terms.

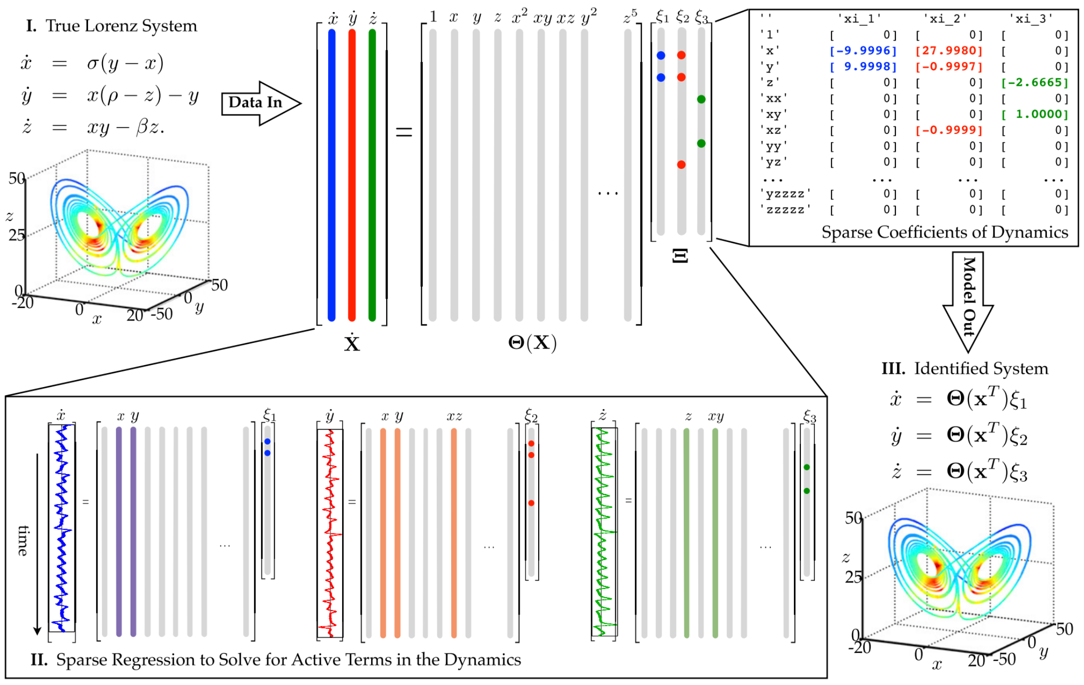
$$\Theta(\mathbf{X}) = \begin{bmatrix} | & | & | & | & & | & | & \\ \mathbf{1} & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \cdots & sin(\mathbf{X}) & cos(\mathbf{X}) & \cdots \\ | & | & | & | & & | & | & \end{bmatrix} \tag{2.17}$$

Here $\mathbf{X}^{P_2}$ and $\mathbf{X}^{P_3}$ are higher polynomials.

Since it is assumed that only a few of the nonlinear candidate terms are relevant for $\mathbf{f}$, the system can be solved as a sparse regression problem. In this way, the sparse vector of coefficients given as $\Xi = [\xi_1 \xi_2 \cdots \xi_n]$, that controls the selection of terms in the solution can be found:

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi \tag{2.18}$$

Figure 2.5 visualize the described process for the SINDy algorithm applied on data collected from the Lorentz equations. $\mathbf{X}$ and its derivative are found from time data from the system. Afterwards, the fewest terms that satisfy $\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi$ is found using a library of nonlinear features $\Theta(\mathbf{X})$. Lastly, sparse regression algorithms are used to find the relevant terms of the right-hand side of the dynamics equations.



I. True Lorenz System
$\dot{x} = \sigma(y - x)$
$\dot{y} = x(\rho - z) - y$
$\dot{z} = xy - \beta z.$
Data In
$\dot{\mathbf{X}}$
$\Theta(\mathbf{X})$
Ξ
Model Out
III. Identified System
$\dot{x} = \mathbf{\Theta}(\mathbf{x}^T)\xi_1$
$\dot{y} = \mathbf{\Theta}(\mathbf{x}^T)\xi_2$
$\dot{z} = \mathbf{\Theta}(\mathbf{x}^T)\xi_3$
time
II. Sparse Regression to Solve for Active Terms in the Dynamics

```
''          'xi_1'      'xi_2'      'xi_3'
'1'       [      0]  [      0]  [      0]
'x'       [-9.9996]  [27.9980]  [      0]
'y'       [ 9.9998]  [-0.9997]  [      0]
'z'       [      0]  [      0]  [-2.6665]
'xx'      [      0]  [      0]  [      0]
'xy'      [      0]  [      0]  [ 1.0000]
'xz'      [      0]  [-0.9999]  [      0]
'yy'      [      0]  [      0]  [      0]
'yz'      [      0]  [      0]  [      0]
 ...          ...         ...         ...
'yzzzz'   [      0]  [      0]  [      0]
'zzzzz'   [      0]  [      0]  [      0]
```
Sparse Coefficients of Dynamics

Figure 2.5: SINDy algorithm applied on data collected from the Lorentz equations. Figure taken from (Brunton et al., 2016).

Parameterized Systems

For many systems, parameters can be used to influence and control the system dynamics, where variations of the parameters can lead to drastic changes in the system (Brunton et al., 2016). Take the Lorenz equation as an example (Lorenz, 1963):

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x)\\ \frac{dy}{dt} &= x(\rho - z) - y\\ \frac{dz}{dt} &= xy - \beta z\end{aligned} \tag{2.19}$$

Here σ, ρ, and β are system parameters. For this particular system, small changes in the parameters can lead to significant changes in the behavior of the system. To consider parameterized systems, the SINDy algorithm can be generalized to allow for analysis of externally forced or controlled systems by adding a feedback control term $\mathbf{u}(\mathrm{t})$ to equation 2.14:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}, \mathbf{u}(t), t). \tag{2.20}$$

2.6 F1 Score

F1 score is a useful metric for evaluating classification prediction tasks, both balanced and imbalanced (Grandini et al., 2020). For imbalanced datasets, the practicality of accuracy as an evaluation metric diminishes because of bias towards the majority class. If a dataset consists of 90% of class A and 10% of class B, then training a model that predicts everything as class A will receive a 90% accuracy. An inaccurate reflection of the quality of the model. In contrast, F1 score measures a more balanced score by considering true positives (TPs), false positives (FPs), and false negatives (FNs) in its evaluation. TPs are the predictions correctly labeled positive, FPs are the predictions wrongly labeled positive, FNs are predictions wrongly labeled negative, and true negatives (TN) are predictions correctly labeled negative. Precision and recall are metrics that express the proportion of positive labels that are actually positive, and the accuracy of the positive label, respectively.

$$Precision = \frac{TP}{TP + FP} \tag{2.21}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.22}$$

The F1 score is defined as the weighted average of precision and recall.

$$F1 - Score = 2 \cdot \left(\frac{precision \cdot recall}{precision + recall} \right) \tag{2.23}$$

However, as seen in equation 2.21, 2.22 and 2.23, the F1 score does not consider TNs. This is because the F1 score is asymmetric and put emphasis on predicting one of the classes correctly rather than correctly predicting all classes. This is useful for the evaluation of cases where the accurate prediction of one class is more important than correctly classifying instances from other classes. F1-score is expressed on a scale from 0 to 1 where a higher score is better.

2.7 Key Python Libraries

In this thesis, python is utilized as the primary programming language. We have employed several Python libraries, among which we highlight two in particular that have been crucial with regard to deep learning and data generation.

2.7.1 PyTorch

For this thesis, we use PyTorch as our deep learning framework. PyTorch is an open-source deep learning framework for Python developed by Facebook's AI research group, and has been publicly available since 2017 (Paszke et al., 2019). PyTorch provides multiple features and applications that can be leveraged to design, optimize, and train a vast range of NN models, and focus on user-friendliness with an API that shares similarities to popular scientific Python libraries such as NumPy and SciPy. This allows for an environment where data processing, model design, and model testing can be done quickly and with much control. In addition, PyTorch provides an array-based programming model accelerated by Graphic Processing Units (GPUs) that can allow for increased training speeds for larger networks. PyTorch simplifies the deep learning workflow by automatically saving parameter values computed during the forwardpropagation in the model training process. Saving the computations along the forwardpropagation allows for increased performance during the backpropagation as these variables can be used directly.

2.7.2 PySINDy

PySINDy is a Python library implementing the SINDy algorithm described in section 2.5 (de Silva et al., 2020). In principle, PySINDy's workflow requires the selection of three components that define a SINDy model. First, a differentiation method is chosen in order to compute $\dot{\mathbf{X}}$ from $\mathbf{X}$. Secondly, a feature library is defined using potential candidate terms that may be a part of the solution. Lastly, an optimization method is chosen for solving the sparse regression problem and finding the sparse vector of coefficients Ξ. After a SINDy object is defined using all three components, the model must be trained on the given data. In the case of parameterized systems, both time history data from the system and the corresponding parameters must be provided. After finding a possible solution, PySINDy allows for evaluation of the solution in the form of scoring the model using test data and provide methods for easily simulating new data using the found equations.

Chapter 3

Methodology and Approach

In this chapter, we cover the approaches and technical work done to execute our experiments. Section 3.1 outlines the dataset pipelines, including processing raw continuous values into discrete values, for the self-generated data. Section 3.2 outlines how OpenLab and the sparse identification of nonlinear dynamics algorithm are used to generate synthetic well-drilling data. Section 3.3 presents the implementation, training, and evaluation of LSTM models. Section 3.4 summarizes the chapter.

3.1 Dataset Generation

In this thesis, we handle a sine wave prediction dataset, a sequence value dataset, and a synthetic well-drilling dataset. All datasets are generated using Python, except for the latter which is obtained from OpenLab's well drilling simulator. Overall, the datasets follow a progression of increasing complexity. In order to make it easier to conduct machine learning tasks with these datasets, a data processing pipeline that prepares the data for model training is implemented. Our models are trained to solve binary classification tasks and require each sequence to have one classification label. In this regard, the dataset structures consist of sequences of varying lengths each with a corresponding binary label denoting whether the sequence is accepted or rejected. All numerical data values in the sequences are discretized into a finite alphabet of symbols, with no duplicate sequences present in the dataset. Overall, the selection of datasets aims to evaluate the rule extraction approach on different structured sequences of continuous values. Datasets have sequences of maximum 30-31 values, ensuring a manageable amount of data, and are also inspired by the sequence lengths used by Weiss et al. in some of their experiments. Additionally, to evaluate extracted DFA and their LSTM models on longer sequences, a testing set consisting of randomly sampled sequences with lengths of 100 is created for all self-generated datasets. We refer to this dataset as "long test data". This section describes each type of dataset, outlines the corresponding construction pipeline, and gives the intention behind utilizing these specific datasets.

3.1.1 Discretization

Discretization is the process of obtaining a nonoverlapping partition of data in a continuous domain by converting numerical data into discrete data consisting of a finite set of intervals. This process enables the mapping of an extensive range of numeric values to a much smaller set of discrete values (García et al., 2013). Prior to rule extraction, the LSTM models must be trained on datasets consisting of a finite set of symbols. This requirement is attributed to the finite and simplistic nature of the DFA unable to represent an infinite set of values. For this reason, datasets of continuous values are first discretized into a finite set of intervals, each mapped to a symbol from a given alphabet.

Notably, a result of discretization is the loss of information depending on the size of the alphabet. Using small alphabets can result in subpar approximations of the original data compared to larger alphabets. In this thesis, we examine the effect of alphabet size on rule extraction and use various sizes for different experiments. Figure 3.1 demonstrates approximations for different discretization intervals. We clearly observe that 25 values retain more information than using 2 values.

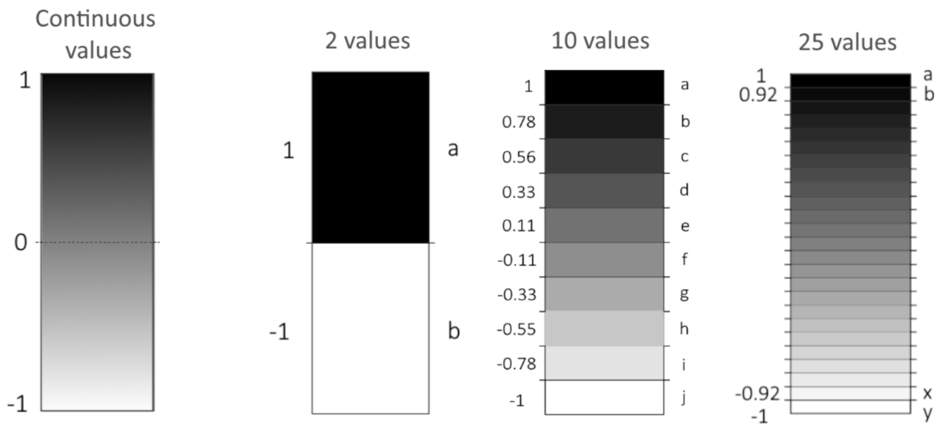


Figure 3.1: Finite set of values discretized from continuous values ranging from -1 to 1, and the corresponding alphabet symbol for each value.

3.1.2 Weights

In binary classification problems, class imbalance occurs when one of the two classification groups contains significantly fewer samples than the other group. For such cases, learners tend to over-classify the majority group, and as a result, instances belonging to the minority group are misclassified more often than those of the majority group (Johnson and Khoshgoftaar, 2019). In an attempt to avoid class imbalance, we follow the approach by Weiss et al. of ensuring that datasets contain close to an equal share of negative and positive data where possible. In some cases, it is not possible to maintain an equal distribution of positive and negative data. For example, the sine wave dataset (section 3.1.3) has very few positive sequences compared to the randomly sampled negative data. On such occasions, class weights are computed to increase the influence of low-quantity classes in the learning process for the models. This is done in PyTorch using the weighted random sampler class when designing dataloaders for the training process using `torch.utils.data.DataLoader`. The weights assigned to the minority class are computed during dataset generation based on the difference in the number of class samples, ensuring that the sum of weights for each class is equal.

3.1.3 Sine Wave Dataset

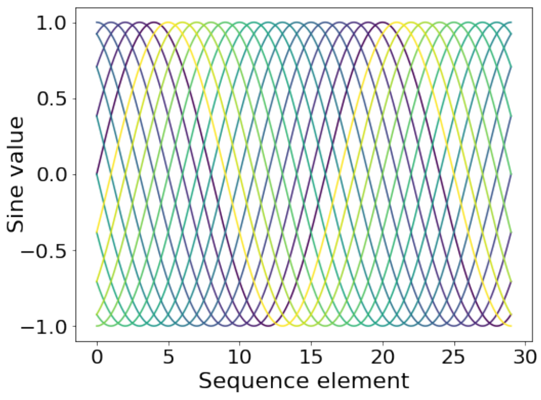
The motivation for using a sine wave prediction dataset is primarily due to its suitability for studying the ability of the rule extraction algorithm to capture repeating patterns with respect to sequence length and alphabet size. Additionally, because of the simplistic and well-understood behavior of sine waves, we can easily validate if we have found an optimal DFA.

Sine wave datasets are created by solving the sine wave equation forward in time, generating several datasets with different angular frequencies and phases. These attributes are controlled by changing ω and ϕ in equation 3.1.

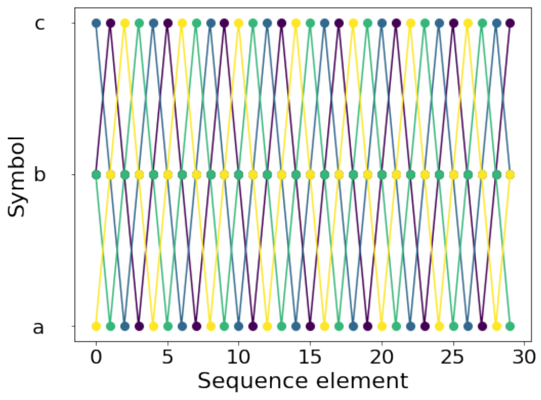
$$y = \sin(\omega t + \phi) \tag{3.1}$$

Here y is the output, and t is the time. The phase ϕ represents the position of the wave relative to $t = 0$.

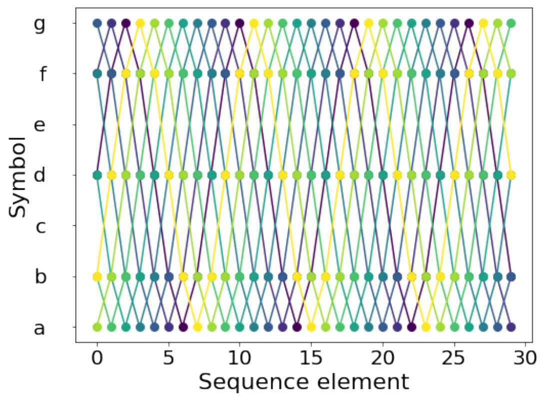
Next, data values are sampled from the sine waves and discretized into an alphabet of symbols, as illustrated in figure 3.2. The resulting sequences of sine wave data points



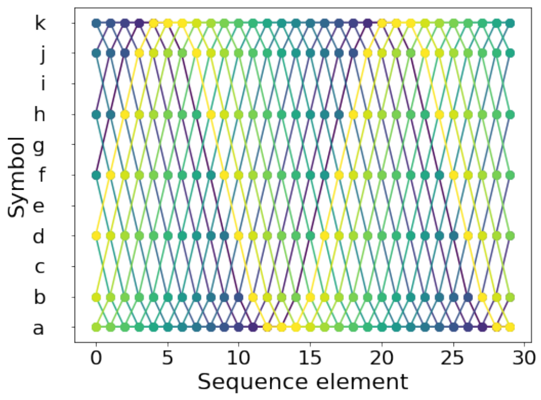
(a) 16 sine waves using continuous values, not discretized.



(b) Sine wave discretized using 3 symbols. Sampled 4 times per period.



(c) Sine wave discretized using 7 symbols. Sampled 8 times per period.



(d) Sine wave discretized using 11 symbols. Sampled 11 times per period.

Figure 3.2: Original sine wave and discretized datasets

define the positive data, while negative data is obtained from sampling random sequences that do not form part of a sine wave. However, as there are fewer combinations of sine wave sequences compared to random sequences, a large discrepancy in the number of positive and negative data samples can be expected depending on the size of the dataset. To address this issue of class imbalance, sample weights are used during model training. Sequences in the dataset are of lengths [1, 2, ..., 29, 30]. We experiment with sine waves sampled 4, 8, and 16 times per period with corresponding alphabets of sizes 3, 7, and 11 respectively. The figure 3.2 illustrates the different sampling rate for each dataset. Note that in representing the waves not all of the symbols are used, as can be seen in figure 3.2c and 3.2d. To optimize the efficiency of model training, we omit the symbols not present in the sine waves from the dataset, and in practice use alphabets of sizes 3, 5, and 7 instead.

3.1.4 Sequence Value Dataset

The intention behind employing the sequence value dataset is to experiment with a controllable simple dataset that in broad strokes emulates the more complex well-drilling dataset by sharing the same structure and sample labeling scheme. Using this dataset, we study the quality of rule extraction on tasks of classifying sequences based on their computed value, and the impact of external influences independent of given inputs on the rule extraction.

We generate a dataset of sequences starting from an initial value and randomly select the next value in the sequence for each time step. The set of possible values represents an alphabet and is defined by discretizing a range of continuous values. For example, a dataset in this manner might employ data values ranging from -1 to 1. These continuous values are discretized down to, for example, 10 intervals with each one corresponding to a symbol in the alphabet. Each sequence has a value for each step in the sequence obtained from the sum of all values up to that specific point. The labeling of each sequence is carried out by assigning a positive or negative label based on whether the value at any point in the sequence exceeds a specified threshold. The labeling scheme is designed to resemble the labeling scheme of the well-drilling dataset described in section 3.2. Figure 3.3 illustrates the value of each sequence for each step for an alphabet of size 10. Sequences are colored either red or blue depending on if the value of the sequences at

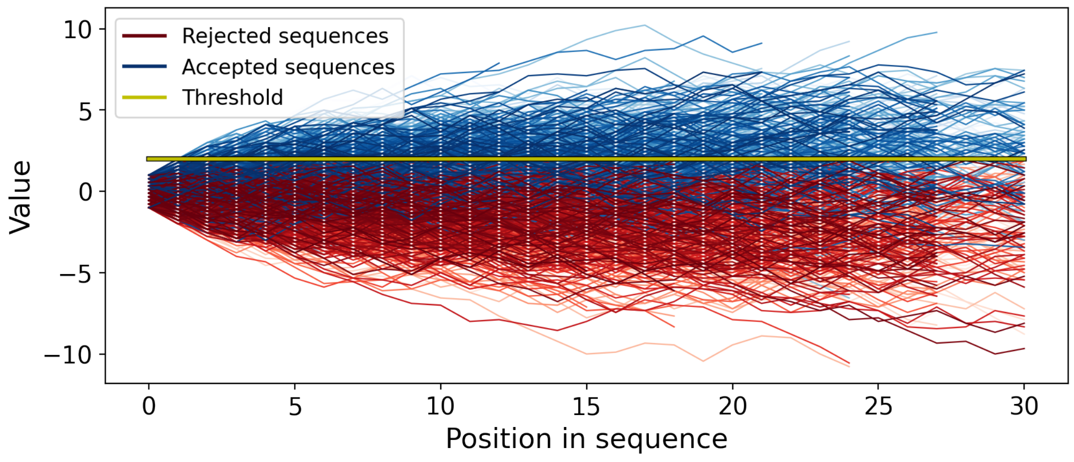


Figure 3.3: The labeling and sequence value of each sequence step for 2000 sequences using an alphabet of size 10. Not all sequences have the same length.

any point is above the given threshold. For this dataset, we experiment with alphabets of sizes 3, 10, and 25.

Additionally, we experiment with an alteration of the dataset that incorporates external factors into the system. Specifically, we introduce an external influence that increases the value of each sequence with 5 after the 10th sequence element, regardless of the given input sequence. The intention behind utilizing this dataset is to examine the performance of the rule extraction approach when progressively incorporating more complexity into the data. For instance, figure 3.4 illustrates a sequence value dataset affected by external influences, regardless of the given input sequence, for an alphabet of size 10. Sequences of lengths [1, 4, 7, ..., 31] are used in both datasets in order to have a more manageable data amount and the ability to analyze longer sequences.

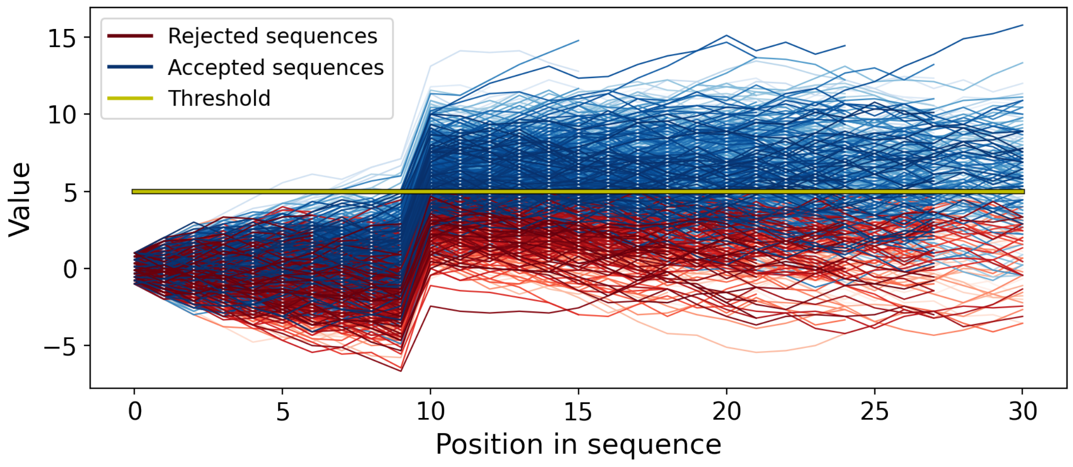


Figure 3.4: The labeling and sequence value of each sequence step for 2000 sequences using an alphabet of size 10. After 10 steps all sequences increase in value.

3.2 Well-Drilling Dataset

The well-drilling dataset aims to measure how well the rule extraction algorithm can be applied to complex physical processes. Specifically, the dataset emulates a drilling scenario where the pressure at the bottom of the well must be maintained below some given pressure threshold. For generating datasets we use a combination of the well-drilling simulator OpenLab and the sparse identification of nonlinear dynamical systems algorithm. We outline our approach for data generation here.

3.2.1 Problem Description

Figure 3.5 illustrates an example of geological constraints that describe the maximum pressure interval allowed for the bottom hole pressure (BHP) of the well down to a depth of 5000m. In this scenario allowing the BHP to exceed or fall below the pressure interval can lead to unstable and dangerous situations for the drilling operation and the drilling crew. Note that although figure 3.5 illustrates a specific pressure constraint for a particular drilling configuration, in many cases, drilling process constraints can be based on multiple aspects, such as safety requirements and drilling equipment limitations. For our case, we do not look at a specific drilling case when selecting a threshold, and instead experiment with an artificial threshold. In order to mimic scenarios constrained by some given threshold, the dataset is composed of input sequences consisting of random setpoint values, labeled according to if the corresponding BHP at any point exceeds the specified

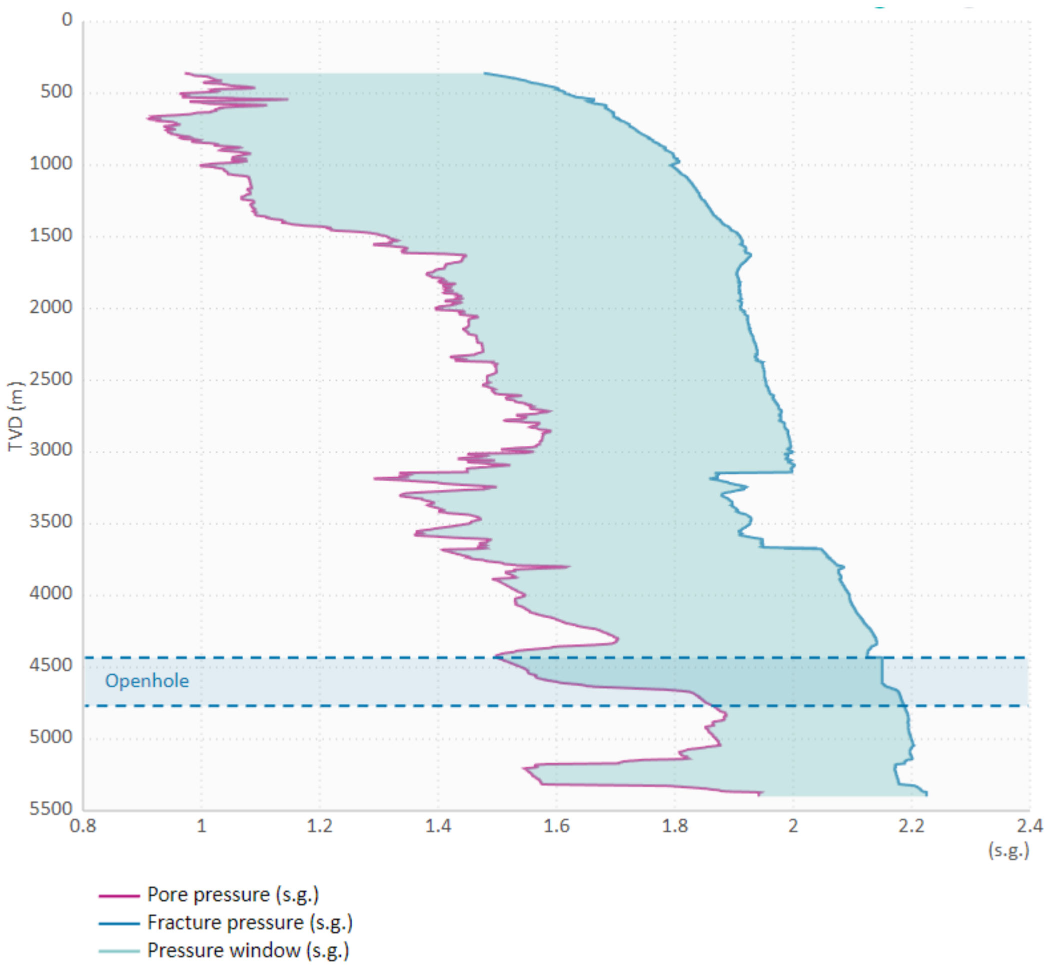


Figure 3.5: Possible geological pressure constraints to a depth of 5000m. s.g. is the specific gravity, tvd is true vertical depth. Screenshot taken from OpenLab (2023a).

threshold. This dataset is in practice similar to the sequence value dataset only more complex due to the involvement of multiple drilling processes.

3.2.2 OpenLab Data Generation

OpenLab is employed to generate realistic well-drilling data. We utilize their web API in Python to extract data and manage simulations. Sequences of setpoints are used to control the simulations. Setpoint sequences are created by randomly sampling values from a predefined set of possible setpoint values. For the simulations, only setpoint parameters such as flow rate and RPM are used as the control inputs, because of their distinct and strong influence on the drilling process. Flow rate and RPM in the context of well-drilling refer to the flow rate out of the main well pump and the rotational speed of the drill string. Moreover, constraining the control inputs to only two input variables results in a smaller and more manageable alphabet. Each combination of setpoint values is mapped to a unique symbol that in turn makes up the alphabet of the dataset. Thus, the size of the alphabet is determined by the product of the number of setpoint values for each parameter. For example, 3 RPM values and 4 flowrate values result in an alphabet of size $3 * 4 = 12$. Figure 3.6 shows a single simulated BHP sequence with the associated inputs of flow rate and RPM used to simulate it.

OpenLab Configuration

To understand this thesis, it is not necessary to have in-depth knowledge about the drilling process. In this work, well drilling is simply used as a prediction problem that needs to be solved. Here we give some details about the simulation settings, for replication of the simulations in OpenLab. For the drilling configuration, we have utilized a generic offshore rig with the default InclinedWell 2500m template. Geology, drilling fluid, drill string, and rig settings are unchanged from the default configuration. During simulations, we use the auto driller in WOB mode, as we experienced some irregularities while using the other implemented auto driller mode, Rate of Penetration (ROP) mode. For the WOB mode, we keep the controller gains as default and set the WOB to 12 tons. Lastly, the simulations use a steady-state torque-drag model with a simulation step of 1 second.

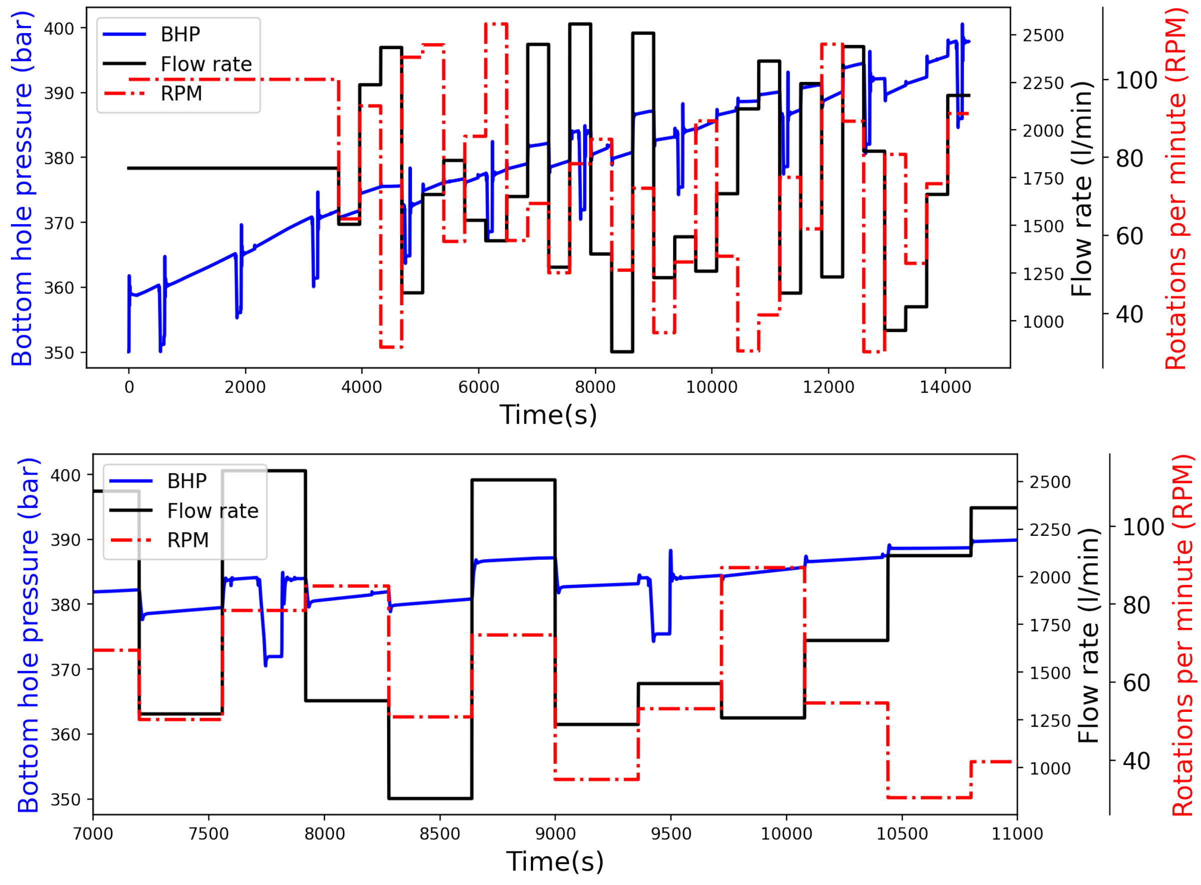


Figure 3.6: Bottom hole pressure versus inputs of flow rate and rpm.

Simulation Design

100 simulations are executed in OpenLab using input sequences consisting of random flow rate and RPM values within intervals of [1000, 2200] l/min and [30, 140] RPM respectively. The parameter intervals are chosen in order to get good coverage of the BHP variations with regard to parameter variation. The sequences have lengths of 30 elements with the simulation changing input every 360th second, using a total of 3 hours to go through all setpoints. Additionally, all simulations run for 1 hour in simulation time with a constant flow rate of 1800 l/min and 100 RPM before initiating the input variation from the setpoint sequences. This approach enables a more comprehensive representation of the dynamics of the well drilling system, by facilitating the build up of physical features such as cuttings, well temperature, and drilling mud properties. Simulation data consists of values recorded for each second of the simulation. To reduce complexity and the total amount of data, we sample every other value of the simulation data.

3.2.3 Synthetic Data Generation using PySINDy

In order to properly train models to generalize on complex dynamics and patterns, a significant amount of data is required. In the case of using the OpenLab simulator for data generation, it became apparent that simulating a vast amount of long simulations is unfeasible with regards to both time and computing power. Generally, simulating 4 hours of drilling can approximately result in 50 minutes of real-world time, moreover, users are restricted to running a maximum of 5 simulations in parallel [1]. These factors result in a very long and tedious data generation procedure if all sequences are generated directly using OpenLab. Also, it would result in unnecessary usage of OpenLabs resources since simulations are computationally heavy to run. In this thesis, model training requires a large amount of data, and generating thousands of sequences of data in this way is impractical.

However, in our case, all simulations are generated using identical well-drilling configurations and are in practice relatively straightforward, seeing as only the change in pressure with respect to the changes in flow rate and RPM is measured. In addition, as illustrated in figure 3.6 the pressure has a fairly simple linear behavior. For these reasons, we employ the SINDy algorithm on a collection of simulations generated by OpenLab in order to find an equation that adequately define the simulation dynamics. SINDy and the Python implementation PySINDy are described in section 2.5 and section 2.7.2 respectively.

Selecting SINDy model

We employ parameterized SINDy with flow rate and RPM as the control parameters together with their derivatives. Utilizing higher derivatives as control parameters allows for better capturing the changes in the system, and we experiment with using derivatives up to and including the 5th-order derivative. During testing, we experience that derivatives computed using smoothed finite difference with a smoothing window of 15 give the best results and use it to compute the derivatives. The PySINDy model is created using a finite difference differentiation method and a feature library consisting of linear terms. This is because we do not have the derivative of the data and must initialize the model with a finite difference differentiation method to numerically compute $\dot{\mathbf{X}}$. It is beneficial to study the data before experimenting with the selection of candidate terms in the feature library.

[1]We have used their academic license

In this case, it is apparent from figure 3.6 that the simulated pressure follows a linear trend, justifying the choice of linear terms in the feature library. For optimization selection, we try out multiple different optimization methods with different hyperparameters and select the model with the lowest mean squared error on the test data. In this case, the best-performing optimization method is the sequentially thresholded least squares algorithm (STLSQ, 2023) with default parameters and a threshold value of 0.0001. Lastly, the chosen model is trained using the generated OpenLab simulation data, and PySINDy gives the equations it finds that best represent the system.

3.2.4 Data Generation and Sequence Labelling

Next, after training the PySINDy model, we use the "simulate" method of the model to simulate the BHP forward in time. The method takes flow rate and RPM setpoint sequences, and an initial starting value as inputs. Flow rate and RPM values are randomly chosen from the intervals [700, 1200, 1700, 2200, 2700] l/min and [20, 45, 70, 95, 120] RPM respectively, and have sequence lengths of [1, 4, 7, ..., 27, 31]. The combination of input parameters results in an alphabet of size 25 for the dataset. The initial value is set to the last BHP value of the 1-hour-long initial drilling part. The PySINDy "simulate" method simulates drilling simulations with durations depending on the length of the input setpoint sequences. Afterward, these simulations are sampled to match the size of the input sequences, in order to get data sequences for our experiments with lengths less than 31. A simulated sequence with its sampling interval is illustrated in figure 3.7.

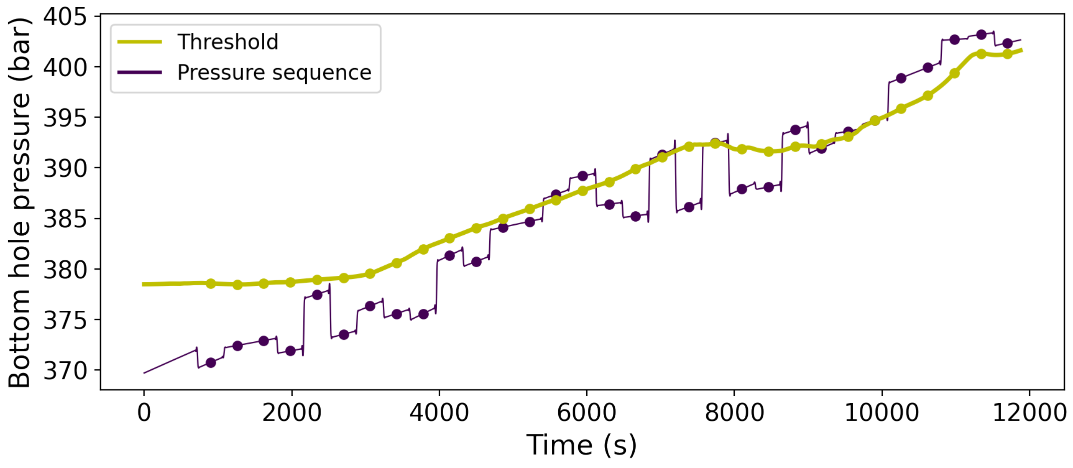


Figure 3.7: SINDy generated sequences based on simulated data from OpenLab. Dots show sampling points.

Sequences are labeled positive or negative depending on if the sequence contains k successive BHP values above the threshold. A $k = 1$ means that if the pressure of the sequences at any point exceeds the specified threshold, the sequence is assigned a positive label. Similarly, $k = 2$ means that two values in a row must be above to receive a positive label. This labeling scheme is based on a drilling scenario where it is possible to exceed the given constraint as long as it occurs only less than some set time. In this case, we have 360 seconds between each setpoint change, meaning that for example in the case of $k = 2$ the pressure can exceed the threshold for 720 seconds before being assigned a negative label.

Figure 3.8 displays a single pressure simulation generated by OpenLab. Over time the pressure increases as the dynamics of the drilling process change. The most notable features are the steep drops in pressure that occur about every 1500 seconds. This is a result of the drilling process ramping down to replace the drill string in order to continue drilling. This process is referred to as auto-connection (OpenLab, 2022). In contrast to prior datasets, we do not use long test data for evaluation. This is because of difficulties with verifying the quality of long sequences generated using the SINDy model given the limited long OpenLab data for comparison.

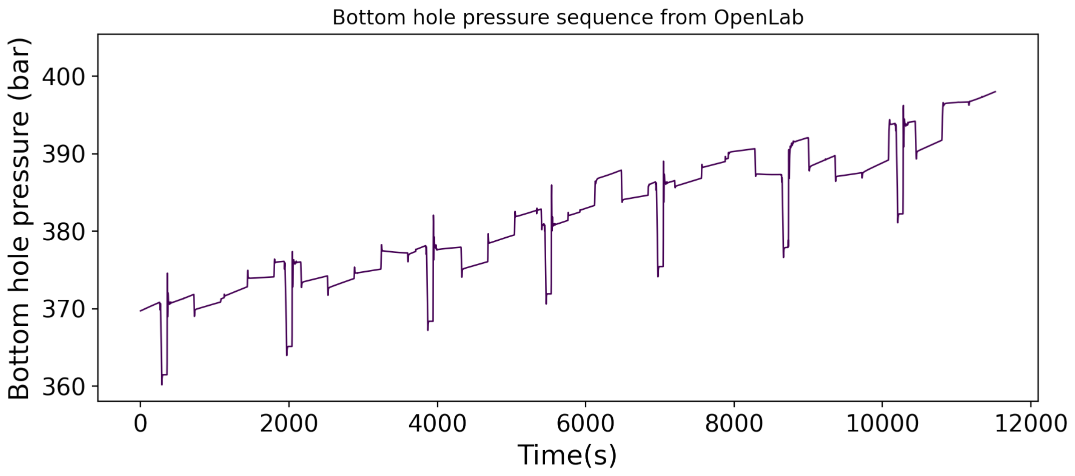


Figure 3.8: SINDy generated sequences based on simulated data from OpenLab. Dots show sampling points.

3.3 Workflow Pipeline

We outline the full workflow for model design, model training, DFA extraction, and evaluation. Python is used as the main programming language for our workflow.

3.3.1 Models

For this study, we utilize LSTM models to solve binary classification tasks. LSTM models are chosen due to being one of the most popular RNN architectures (Murdoch and Szlam, 2017), and in addition, are used by Weiss et al. (2018) in their experiments. The LSTM models in this study are implemented using the open-source machine learning framework PyTorch. To create the LSTM models we extend the PyTorch's `torch.nn.Module` to make a generic and configurable model design. The code for the model is given in appendix A.1. The model design is inspired by examples from the PyTorch code base PyTorch (a) and PyTorch (b). The code contains extraction methods used for extracting DFA, explained in 3.3.3. Preliminary experimentation showed that using an embedding layer of size 50 gave satisfactory results with regards to training time and extraction, leading us to it in our implementation. Additionally, we use a dropout of 30% in order to reduce overfitting and improve on generalization.

Some data processing is performed between layers and for each forward pass of the model:

- Between each forward pass, the hidden states and cell states are reset to zeros. This is because we want the training of each sequence to be independent.
- As a result of working with sequences of varying lengths, the sequence lengths are standardized to the length of the longest sample in the dataset using padding. This allows for batching sequences of different lengths together.
- After embedding, we format the sequences for the LSTM layer to ignore the padded values, using PyTorch's pack_pad_sequence function. Packing the sequence indicates what values are padding for the LSTM model and omits them during training. This optimizes the computation and eliminates concerns about the padded values affecting the result. Afterward, the output of the LSTM layer is unpacked and passed to the next layer.

3.3.2 Training Script

A script has been created for training the LSTM models which utilizes weighted data samples from the dataset generation and processing pipeline. Sample weights, training data, and batching is handled using PyTorch's `torch.utils.data.DataLoader` class.

Backpropagation and updating model weights are handled by PyTorch during training. PyTorch offers a wide range of optimization methods and loss functions to choose from. For this thesis, the Nadam optimization algorithm (Dozat, 2016) and the Binary Cross Entropy (BCE) loss function are chosen. Specifically, we use PyTorch's `torch.nn.BCEWithLogitsLoss` loss function that combines a sigmoid and BCE loss function. For this reason, we omit the sigmoid layer in our models during training. For the Nadam optimization algorithm, we use default hyperparameters and a learning rate of 0.002

Before training, we do some necessary preprocessing of the data. After discretization, each value is mapped to its corresponding alphabet symbol, and any duplicate sequences are removed. Furthermore, each sequence is weighted and labeled based on class distribution and the dataset labeling scheme respectively. Regarding the sequence value dataset and the well-drilling dataset, sequences exceeding the threshold are labeled specifically as positive. This is because the F1 score focuses on predicting true labels, and in these experiments, we want to correctly predict these sequences, given their importance in our safety-critical problems. For each dataset, the data samples are split into two parts: a training split and a test split. The test split provides an unbiased evaluation of the models and is kept separate from the training process and any data processing before evaluation. In order to reduce training time, we leverage PyTorch's support of hardware acceleration by putting tensors on the GPU (Paszke et al., 2019).

3.3.3 DFA extraction

Accompanying their paper, Weiss et al, provide an implementation of their rule extraction approach in Python at a publicly available code base on `github.com`. Their implementation allows for DFA extraction on any networks that implement their rule extraction API consisting of three methods (Extraction, 2022):

1. **classify_word():** A method for classifying an input word as either True or False.
2. **get_first_RState():** A method that returns the initial state of the network and whether the state is rejecting or accepting.
3. **get_next_RState():** A method that takes a state and a symbol as input, and returns both the new state after processing the symbol and if the new state is rejecting or accepting.

4. Lastly, we include an instance list containing the symbols of the alphabet, necessary for performing the extraction[2].

The states referred to above, in the case of LSTM networks, are formed by concatenating the hidden state and the cell state of the network. For batch processing, the last entry of the batch matrix containing the hidden states and the cell states is used. For multi-layered LSTM networks, the state of each layer is concatenated to represent the state of the entire network. Consequently, when processing an input, the input state must first be split into the hidden state and the cell state, due to the LSTM layers using these states as its initial states. Afterward, the new hidden and cell states obtained from the LSTM layer are concatenated and returned as the new state. Additionally, recall that we do not use a sigmoid layer in our model design as this is handled during training by the `torch.nn.BCEWithLogitsLoss()` loss function. For this reason, we pass each output through a sigmoid layer before classifying the input. Appendix A.1 displays our implementation of the L* extraction methods for an LSTM network. To our knowledge, this is the first PyTorch implementation of Weiss et al.'s rule extraction methods.

Target Examples Before each extraction, examples of at least one accepting, and one rejecting sequence must be provided. We find starting examples using a self-created function for searching through the training dataset for samples of wanted lengths. We use the implemented method for `classify_word()` of the trained LSTM model to find accepting and rejecting examples. This means that the examples in practice are not guaranteed to be correctly labeled, as the trained model decides the classification. If no examples of accepting or rejecting sequences are found for the given length, it searches for a sequence one unit longer until at least one accepting, and one rejecting sequence is found. We experiment with the effect of different-sized target examples in section 4.3.3.

Time Limit and Refinement Depth The extraction method also supports setting a max time limit and an initial refinement depth. If the extraction uses longer than the set time limit, the extraction terminates and returns the last proposed DFA at that time. The initial refinement depth decides the initial partitioning of the state space, which helps to prevent premature termination of the extraction process and reduces the likelihood of extracting small inaccurate DFA, albeit at the cost of longer running times. Refinement depth is described in more detail in section 4.3.2. The extraction time and refinement depth are given for each experiment in the next chapter 4.

[2]This list is not mentioned by the authors, but is essential for the extraction process.

3.3.4 Evaluation Script

In order to perform an assessment of our models, we create a script that measures the performance of the LSTM models and their extracted DFA over different evaluation data. Evaluations are conducted using multiple datasets, depending on the experiment, such as training, test, and long test data. This script utilizes the data pipeline outlined in the previous sections for generating training and test data. The model performances are evaluated using the F1 metric. Furthermore, we measure the total inference time for all models by summing together all prediction times during evaluation. Additionally, we pass the output through a sigmoid layer before classification.

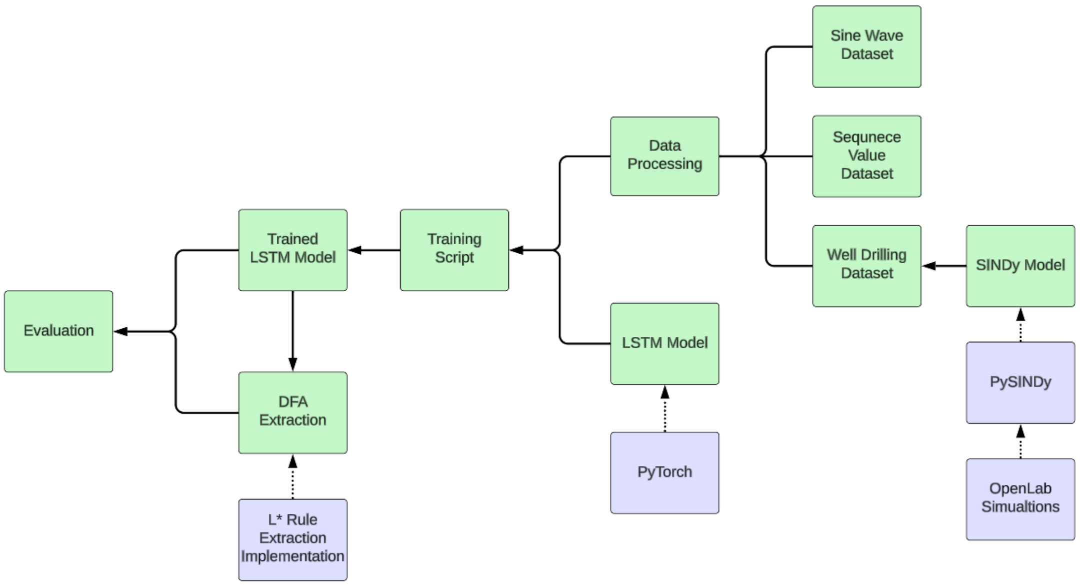


Figure 3.9: Workflow with external resources.

Figure 3.9 provides an overview of the full workflow, encompassing dataset generation, model training, rule extraction, and evaluation. Additionally, it showcases the integration of self-generated code and external resources. Green boxes indicate self-made code, while blue boxes are external resources utilized in the workflow.

3.4 Summary

Creating the datasets and models used to present our final results involves substantial overhead, as is typical with most machine-learning problems. To address these challenges, multiple dataset pipelines have been developed that streamline the process of generating positive and negative sequences of varying lengths, continuous data values are discretized into alphabets of symbols of fixed sizes, and sample weights have been computed to mitigate class imbalance. Moreover, in addition to self-generated data, we leverage the synthetic well-drilling simulator OpenLab together with the SINDy algorithm to conduct evaluations of the rule extraction approach on models trained on synthetic physical processes. To facilitate reproducibility and transparency, we provide a workflow pipeline for our experiments, which includes details about our data generation, model design, training process, DFA extraction, and evaluation procedures.

Chapter 4

Experiments and Results

This chapter presents the obtained results from carrying out rule extraction on multiple test cases, as well as conducting an examination of the rule extraction algorithm to provide deeper insight into its capabilities and limitations. Experiments are structured to progressively increase in complexity, culminating with the well-drilling dataset. For some evaluations, we utilize test data that is longer than the sequences used for training. These sequences are referred to as long test data and are of length 100. The illustrated DFA graphs are created from Weiss et al.'s rule extraction code implementation.

4.1 Experimental Settings

For this study, the methods and datasets have all been implemented in Python 3.9.15. All code has been developed and executed in Windows Subsystem for Linux (WSL) 2 kernel version 5.10.102.1 running in a Windows 10 system. WSL enables running Linux environments on Windows machines without using separate virtual machines or dual booting (Microsoft, 2023). The DFA extraction has been implemented using the code repository for L* rule extraction provided by Weiss et al. on github.com (Extraction, 2022). The RNN models have been constructed using PyTorch 1.12.1. OpenLabs python API version 2.5.6 have been used for drilling control and extraction of simulation data from their well-drilling simulator (OpenLab, 2023a), using an academic license. PySINDy version 1.7.3.dev163+g5c6e9fd have been used to generate synthetic data. The LSTM models have been trained using Nvidia's RTX2070 GPU, AMD Ryzen 7 2700X, and 32GB of RAM.

4.2 Evaluation Metrics

To assess the quality of extracted DFA, comparisons are made with the model employed for extraction. Most datasets used in our experiments contain an uneven amount of positive and negative data samples. Consequently, to mitigate the impact of class imbalance, weights are used during training and F1 score is used as an evaluation metric.

4.3 Extraction Parameters

We give a description of the parameters that affect the rule extraction algorithm. The application of the rule extraction approach in this thesis is different from Weiss et al. in their work. Mostly, in their work, the focus is on finding the smallest possible DFA that accurately represents the RNN, known as a minimal DFA, on some regular languages. In this thesis, we examine the capabilities of the method for more difficult problems that are not expected to be perfectly represented using small DFA. For these reasons, we deemed it necessary to perform some initial experiments, evaluating the influence of parameter selection on the quality of extractions for the more complex datasets employed in this thesis. The parameters we explore are described as follows:

4.3.1 Extraction Time

In cases where the DFA extraction process is unable to reach equivalence on a language identical to the language trained on by the RNN, longer extraction time gives the algorithm more time for finding larger and potentially more accurate DFA. Optimally, we want to find small DFA that capture the behavior of the RNN accurately. If this proves unattainable, allowing the algorithm more time to find larger DFA may yield better results, or on the contrary may lead to an overfitted DFA. By experimenting with various extraction time limits, we evaluate the influence of extraction time on extraction quality for a test case where the algorithm is unable to converge to a DFA solution.

4.3.2 Initial Refinement Depth

The initial partitioning and refinement operations of the abstraction are very coarse, making the method prone to accepting very small and inaccurate DFAs (Weiss et al., 2018). To prevent early termination into small and inaccurate DFA, an initial refinement depth is chosen prior to running the algorithm. The refinement depth decides the initial partitioning of the state space and can be increased if there is a suspicion that the extraction terminates too early. However, choosing a too-large refinement depth can make the abstraction too large for practical exploration, and make the total extraction process very time-consuming. We explore the influence of refinement depth on DFA extractions from LSTM models trained on complex datasets, and make comparisons with different-sized starting examples.

4.3.3 Starting Examples

We study the influence of different-sized starting examples on the extraction processes. To assist the extraction process, positive and negative starting examples can be supplied prior to extraction. Weiss et al. in their work, employ the shortest possible starting examples for their experiments with the aim to obtain the smallest possible DFA that accurately represents the RNN. Nevertheless, some datasets are complex making it difficult or impossible to accurately represent the model using small DFA. Terminating early for such cases can lead to extracting inaccurate DFA. Using longer starting examples for our experiments may help the extraction process avoid early termination and find more accurate DFA by expanding the initial abstraction.

4.3.4 Effect of Extraction Parameters

This section presents the performance of extracted DFA for different extraction times, refinement depths, and starting example lengths. All DFA are extracted from LSTM models trained on a sequence value dataset (outlined in section 3.1.4) with 2000 random samples per sequence length, and an alphabet of size 10. Sequences are of lengths [1, 4, 7, ... 31], the LSTM model has 2 layers and 64 nodes, a 30% dropout, and uses a batch size of 64 for the training. Evaluations use LSTM training data, test data, and 10 000 sequences of long test data. Throughout the tests, all extraction parameters are kept consistent except for the specific parameter being examined. If not stated otherwise, extraction time for all cases is 600 seconds, refinement depth is 10, and starting example length is 1. Results are averaged over 3 runs with different starting examples.

Extraction time (s)	5	60	600	LSTM model
F1 score train data	0.668	0.824	0.824	1.0
F1 score test data	0.662	0.814	0.814	0.977
F1 score long test data	0.665	0.842	0.842	0.974

Table 4.1: F1 scores of DFA and LSTM model for different extraction times.

Refinement depth	5	10	15	LSTM model
F1 score train data	0.867	0.829	0.824	1.0
F1 score test data	0.853	0.823	0.816	0.977
F1 score long test data	0.853	0.850	0.847	0.974

Table 4.2: F1 scores of DFA and LSTM model for different refinement depths.

Starting example length	1	10	22	LSTM model
F1 score train data	0.715	0.848	0.515	1.0
F1 score test data	0.721	0.842	0.516	0.977
F1 score long test data	0.823	0.854	0.540	0.974

Table 4.3: F1 scores of DFA and LSTM model for different length starting examples.

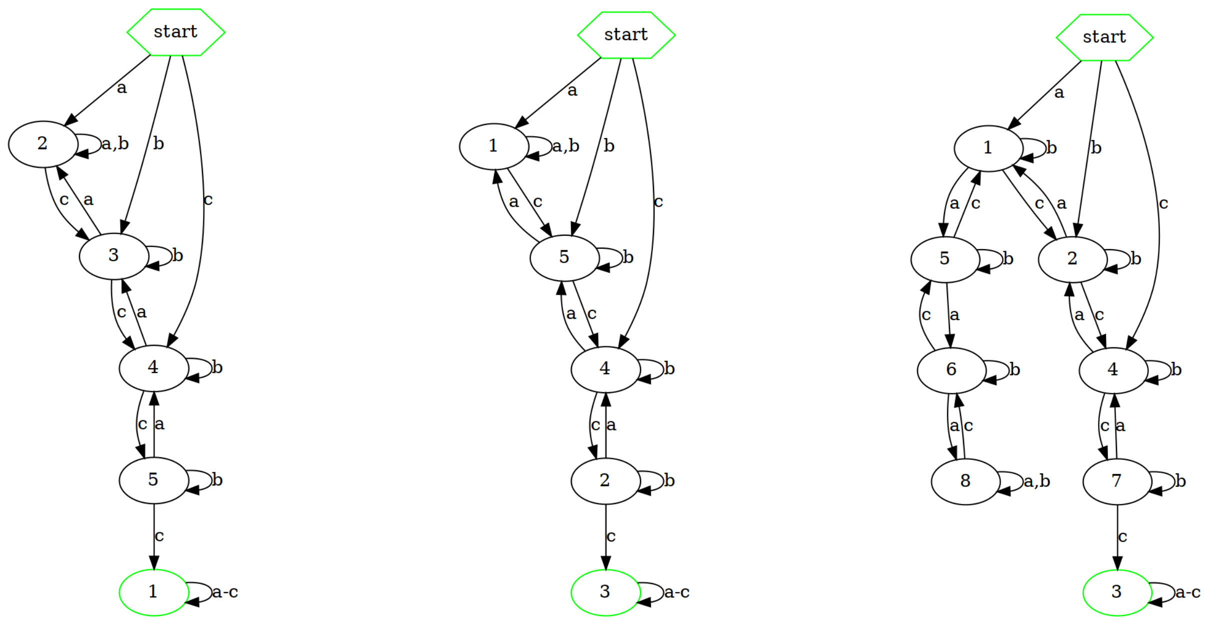
Extraction Time Table 4.1 show the F1 score for DFA extracted using different extraction times. All DFA perform worse than their respective LSTM models. Extraction times of 60 and 600 seconds have found the same DFA, and we see no difference in performance between them. This is most likely due to the complexity of the dataset used for experimentation, resulting in the algorithm being unable to build up a better DFA within the time limit after the first 60 seconds. This is possibly due to the creation of a large abstraction resulting in the algorithm not being able to finish refinements before the time limit. Despite this we see a clear improvement over the 5 second time limit for both cases. This indicates that higher extraction times might improve on the quality of rule extraction on this type of dataset.

Refinement Depth Moreover, we see in table 4.2 that changes in refinement depth for this test case do not significantly improve on the extracted DFA with regard to F1 scores. All cases end up building very similar performing DFA, with a refinement depth of 5 scoring a bit higher than the rest. No extractions terminate early. However, we note that increased refinement depth results in more time used to find the initial partition, resulting in less time for building up a DFA within the given time constraint. Refinement depths of 5 and 10 have negligible impact on the processing time, while in contrast, a refinement depth of 15 uses ∼50 seconds longer for the first abstraction for this test case

[1].

Starting Example Length Table 4.3 shows that longer starting examples can have a positive and negative influence on the F1 scores of the extracted DFA. The DFA extracted using a starting example length of 10 outperforms all other DFA, while on the other hand, using a length of 22 gives the lowest average F1 scores. The extracted DFA using length 22 has most likely ended up with a very large abstraction, that it is unable to explore and refine within the given time constraints. This suggests that simply increasing the starting example length does not necessarily result in better performance.

To visualize the possible performance increase of using longer starting examples for datasets of this type, the same test can be conducted using an alphabet of size 3. Figure 4.1a and 4.1c demonstrates the difference in extracted DFA for starting examples of length 1 and 15 respectively. Here green nodes represent accepted states while black nodes are rejected states. The graph shows transitions between states depending on the next symbol. The numbers are used to identify states. We see that the smaller DFA terminates too early, and as a result, in this case, receives a lower F1 score compared to the extraction using a longer starting example. Furthermore, in this scenario, increasing



(a) Starting example length of 1

(b) Refinement depth 15

(c) Starting example length of 15

Figure 4.1: Extracted DFA using an alphabet of size 3.

[1]The initial refinement time is dependent on the dataset. Using an alphabet of size 3 results in about 20 seconds longer initial refinement time with a refinement depth of 15.

refinement depth to 15 and using a starting example length of 1 does not provide any improvements as illustrated in figure 4.1b.

Summary

It is likely that all DFA extractions performed in these experiments suffer from the limited extraction time and are unable to find larger and better DFA. We observe that increasing starting example length can help with finding better DFA within the time constraints for these particular experiments. However, this is not guaranteed and can result in lower-quality extractions. We also observe that increasing refinement depth does not provide any noticeable performance increase for the extractions in our test cases. Overall, we remark that in order to most likely improve on DFA extractions for datasets in this thesis, we should use longer extraction times, and be reserved in the selection of starting example lengths.

4.4 Method Limitations

The rule extraction approach, as described in section 2.3.3, employs a minimally adequate teacher to answer queries in order to build a DFA from the ground up. Here the minimal adequate teacher is a trained LSTM model, and as a result, the speed at which the model is able to answer queries will influence the extraction time, especially for cases with many queries. The inference time for a NN model depends on the size and the complexity of the model, with larger networks requiring more computations for producing an output. For cases where the rule extraction is able to converge to a small DFA, extraction time does not pose a significant problem, as relatively few queries are needed. On the other hand, for cases where the algorithm fails to converge to a solution, and indefinitely builds up a DFA, the inference time of the model can affect how large DFA can be built within a given time interval. For scenarios where the extraction process is restricted to a set time limit, this might influence the quality of the extracted DFA. Thus, requiring some thought when considering the complexity of the model for problems that are bound by extraction time limitations. Additionally, we investigate the influence of DFA size on prediction quality for our experiments.

In order to gain insight into the correlation of model size and extraction time, we conduct evaluations of DFA extracted from multiple LSTM models of various complexity.

Prior to rule extraction, the models were trained on a small dataset similar to the sequence value dataset 3.1.4 with 500 random samples for each of the sequence lengths [1, 4, 7, ..., 31], until an F1 score of 1.0 was reached. Extraction uses starting example length of 1, an extraction time of 120 seconds, and a refinement depth of 10. Results are averaged over 3 runs using different starting examples.

Results

Hidden units	16	64	256	16	64	256
Layers	1	1	1	2	2	2
LSTM F1 score on test data	0.981	0.978	0.956	0.970	0.976	0.966
F1 score on test data	0.827	0.740	0.733	0.871	0.733	0.806
DFA node count	276.67	250.67	115	84.67	50.67	115

Table 4.4: Average number of nodes for extracted DFA and resulting F1 score on the test dataset.

Table 4.4 shows the average number of nodes for the DFA extracted from LSTM models of different complexity, and the corresponding F1 score on test data. For this experiment, none of the extractions converge to a representative DFA. We see that regardless of model complexity, all LSTM models score very high on the test data, with scores in the high 0.9. Furthermore, we see that with increasing complexity comes a decrease in DFA size, except for the most complex LSTM model with 2 layers and 256 nodes extracting a DFA with a node count of 115. In general, models with 2 layers produce lower node counts. Nevertheless, from the presented data there seems to be little correlation between the number of nodes in the DFA and the achieved F1 score. For example, the two best scoring DFA, extracted from LSTM models of 1 layer 16 nodes and 2 layers 16 nodes, have each an average node count of 276.67 and 84.67, but scores very similar. Another example is the large difference in F1 score between the two DFA with the highest average node counts of 276.67 and 250.67, with a score difference of about 0.1, extracted from LSTM models of 1 layer 16 nodes and 2 layers 64 nodes. Overall, we observe some varying influences of model complexity on the size and the performance of the DFA. However, for this test case, we can not conclude that extracting larger DFA directly results in better predictions. For this reason, when conducting experiments on datasets of this type, such as the sequence value dataset with external influences (section4.5.2) and the well-drilling dataset (section 4.5.2), we refrain from restricting ourself to using very small LSTM models.

4.5 Test Case Results for Self Generated Datasets

In this section we evaluate the performance of extracted DFA on the two self-generated datasets, the sine wave dataset and the sequence value dataset, described in section 3.1.

4.5.1 Sine Wave Dataset

Models trained on the sine wave dataset predict whether a given sequence is fully part of a sine wave. This experiment is the simplest of all the test cases, and is used as a reference for the more complex datasets. Three datasets are created by discretizing full sine waves using alphabets of size 3, 5, and 7, as illustrated in figure 3.2. The sequences are of lengths [1, 2, ... 30], with the sine sequences labeled as the positive data. For each sequence length, 1000 random samples make up the negative data. There are very few sequences of sine waves compared to the sequences of negative data resulting in a class imbalance. The class imbalance is handled using weighted samples during training. One LSTM model using 2 layers with 64 nodes and a dropout of 30% is trained on each dataset. The models are trained until an F1 score of 1.0 is reached and use a batch size of 64 samples. For this experiment, we evaluate how well the algorithm finds patterns and minimal DFA in the data, and for these reasons employ small starting examples of length 1 and a maximum extraction time of 120 seconds for the rule extraction. This is because, longer starting examples can make it difficult to find a minimal DFA, and a long extraction time is not important when we want to find small DFA. The refinement depth is 10, and results are averaged over 3 runs. Moreover, for these test cases, we evaluate performance on training data and 10 000 samples of long test data. Regular length test data is not used since there are few positive samples and removing some from the training data to use as test data might negatively affect model learning. Additionally, this evaluation is helpful in finding any obvious flaws in our implementation, since we know the behavior of sine waves and can use it for validating extracted DFA.

Results

Figure 4.2 illustrates an extracted DFA on the dataset with alphabet size 3. The algorithm perfectly captures the full sine sequence for each starting symbol and loops back on itself. Green nodes represent accepted states while black nodes are rejected states. The graph shows transitions between states depending on the next symbol. Comparing the minimal

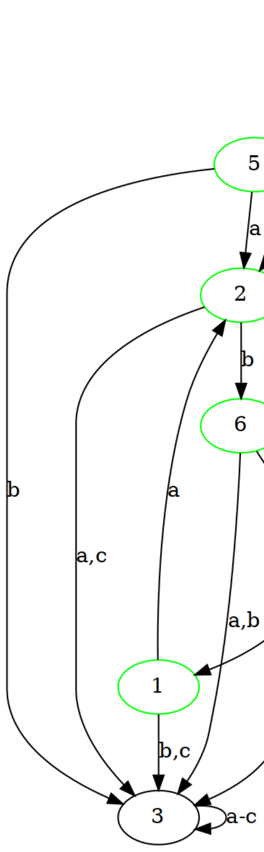


Figure 4.2: Correct DFA extracted from 4.5.1. Illustrates the ground truth of the sine dataset using an alphabet size of 3.

DFA for alphabets 3 and 5 using figure 4.2 and 4.3, we observe the impact of increased complexity in the DFA. The DFA with an alphabet size of 5 has more possible transitions for each state, and a longer sequence is needed to loop over one wave period. Table 4.5 presents comparisons of multiple extracted DFA using different-sized alphabets and their respective LSTM models, on training and long test data. The results show that all LSTM models score perfect on both datasets, which is also the case for all extractions able to find a minimal DFA, such as DFA1 and DFA2. Despite not reaching a solution within the specified extraction time constraint, DFA3 using an alphabet of size 7, is able to achieve a very high score on both training and long test data. Only a single misclassification is made for all of the three extractions. Furthermore, this dataset has a higher sampling rate and needs 16 datapoints to complete a wave period. In contrast, the smaller alphabet datasets need only 4 and 8. This suggests it is harder for LSTM networks to generalize and learn longer sequences well enough for the rule extraction to converge to a DFA.

Despite scoring perfect on the evaluation metrics, we observe that the LSTM models are still capable of making wrong predictions and are susceptible to adversarial examples. Adversarial examples are inputs that are indistinguishable from real samples and are

Model	Alphabet size	F1 score train data	F1 score long test data	DFA nodes	Converged
LSTM1	3	1.0	1.0		
DFA1	3	1.0	1.0	4	Yes
LSTM2	5	1.0	1.0		
DFA2	5	1.0	1.0	13	Yes
LSTM3	7	1.0	1.0		
DFA3	7	0.989	0.989	195.3	No

Table 4.5: Performance of models and extracted DFA for the sine dataset.

classified incorrectly by the network (Madry et al., 2018). This observation is also made by Weiss et al. in their work. For this test case, we notice that minor changes in a sine wave sequence are enough to be incorrectly predicted as true and see that such inaccuracies can be inherited by the extracted DFA. Figure 4.3 and figure 4.4 show the DFA graphs for two extractions, one using a short starting example of length 1 and one using a longer starting example of length 7. Both extracted DFA achieve a perfect F1 score on the training data similar to the LSTM model used for extraction. However, we see that figure 4.4 does not represent a perfect sine cycle. The most noticeable example of a wrongly predicted sequence is the rejected state 7 between states 4 and 10. The acceptance criteria require that the sequences are fully part of a sine wave, meaning that no rejected states should be present between two accepted states. Inserting the state path sequence passing through state 7 into the LSTM model results in an incorrect positive prediction. This highlights the importance of model generalization, as despite the LSTM model being trained to predict all the training data correctly, model imperfections were passed through to the DFA during the extraction process. Furthermore, this particular comparison also demonstrates how shorter starting examples can be better for converging and finding minimal DFA.

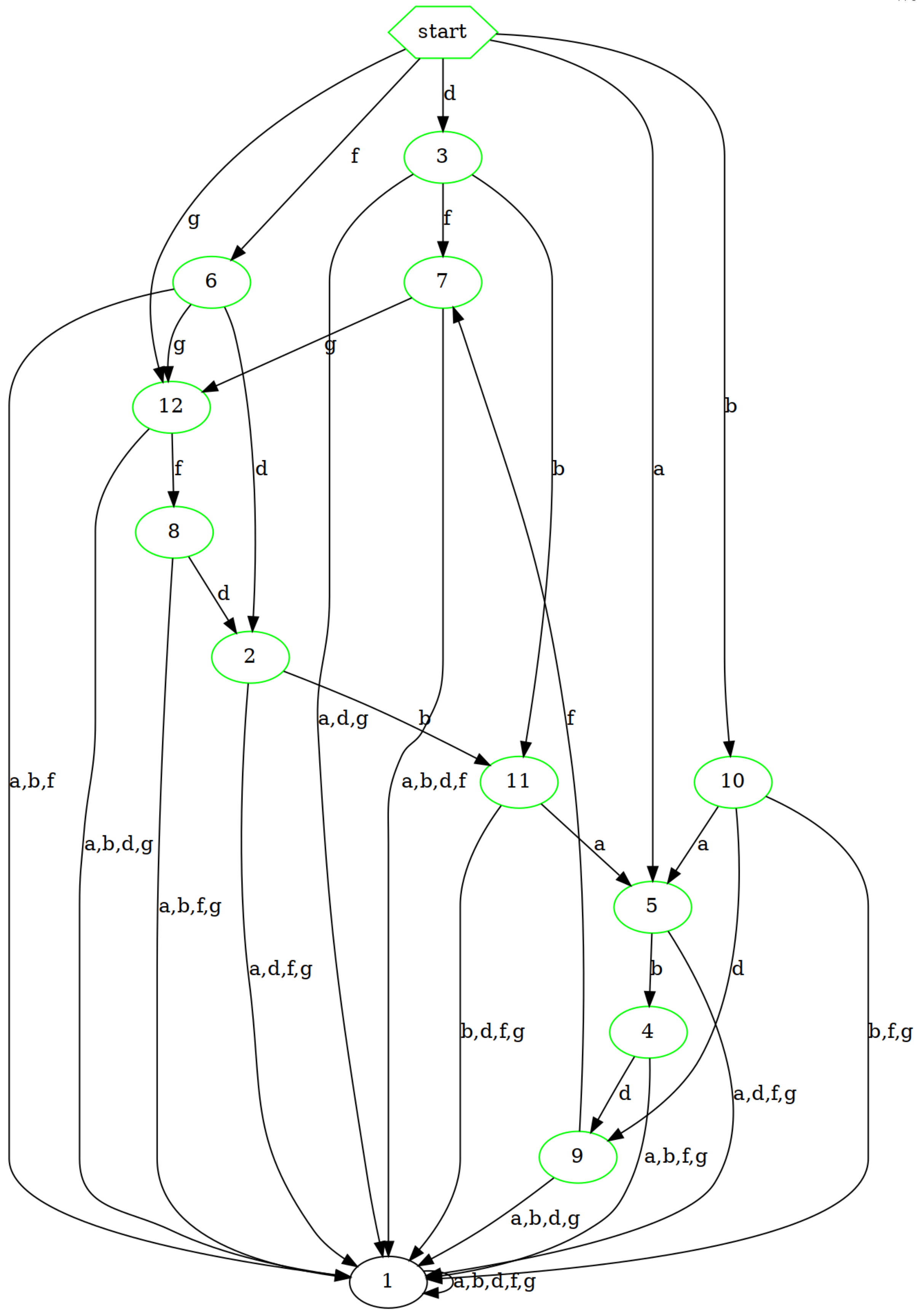


Figure 4.3: Correct DFA extracted from 4.5.1. Only sequences that form a part of a sine wave are accepted. This illustrates the ground truth of the sine dataset using an alphabet size of 5.

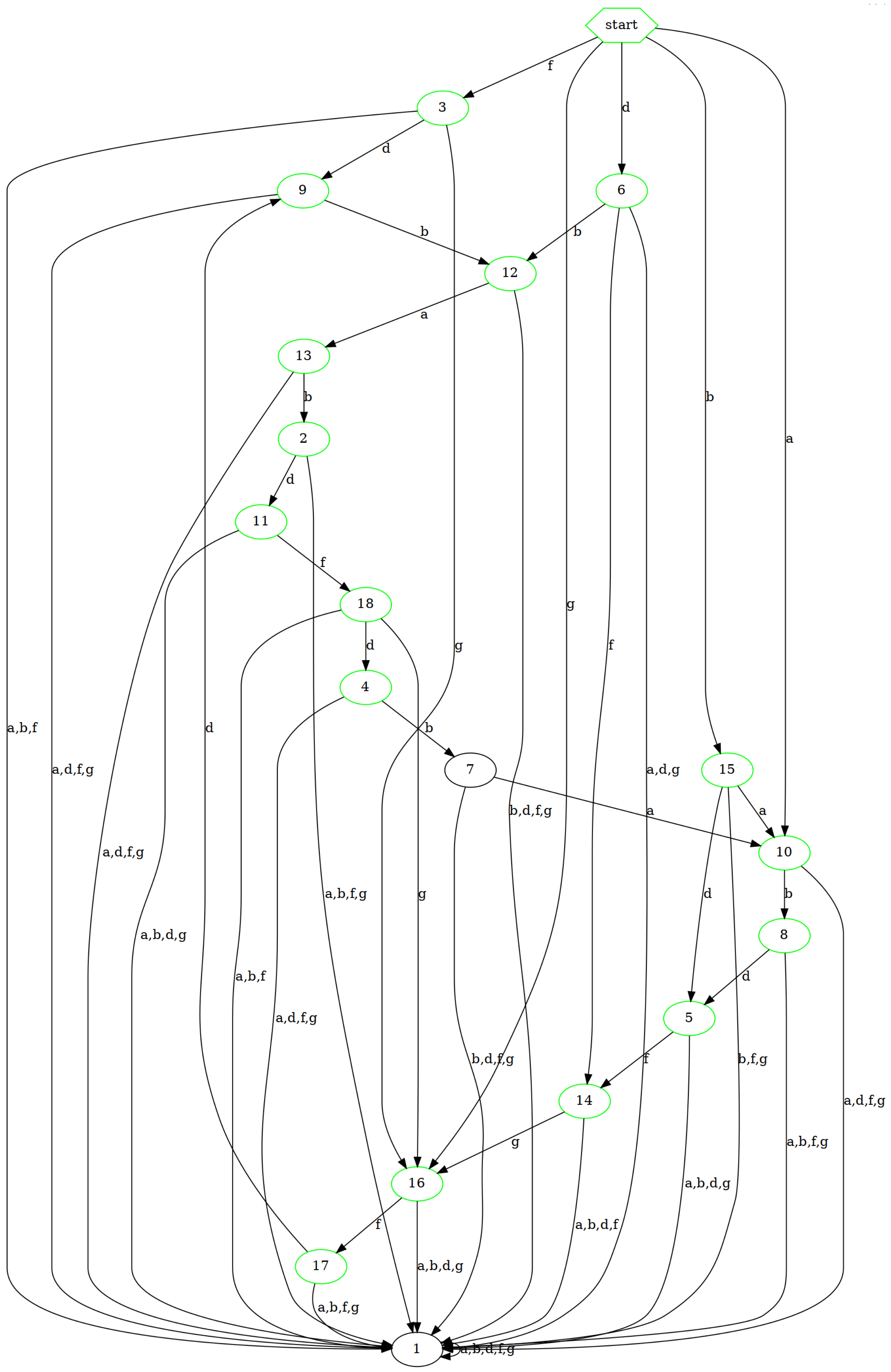


Figure 4.4: Incorrect DFA extracted from 4.5.1.

4.5.2 Sequence Value Dataset

In the previous section, we observed that it is possible to find minimal DFA for a simple problem. Following this, we evaluate the approach on more complex data. We evaluate the performance of rule extraction on LSTM models trained to predict if the "value" of a sequence at any point exceeds a specified threshold. This test case aims to experiment with increased data complexity, as well as measure the rule extraction algorithm's ability to handle external influences affecting the data independently of the input. For this reason, we perform two test cases that are designed with or without the effect of external influences.

Three LSTM models are trained on datasets with alphabets of size 3, 10, and 25. The sequences using alphabets of size 3 and 25 can be seen in figure 4.5 and 4.6, all values above the yellow line are accepted and values below are rejected. These figures

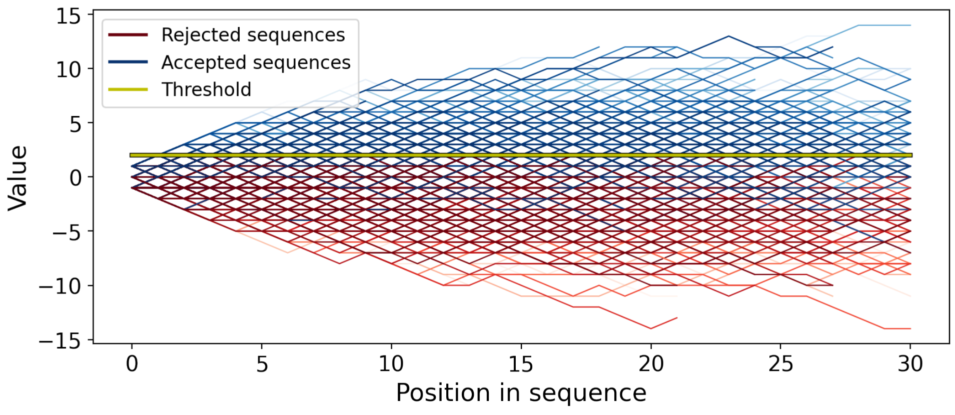


Figure 4.5: 2000 sequences using an alphabet of size 3.

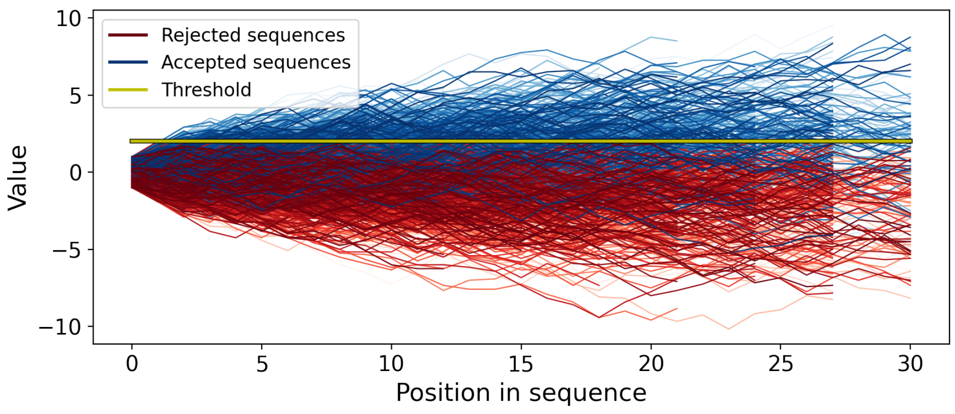


Figure 4.6: 2000 sequence using an alphabet of size 25.

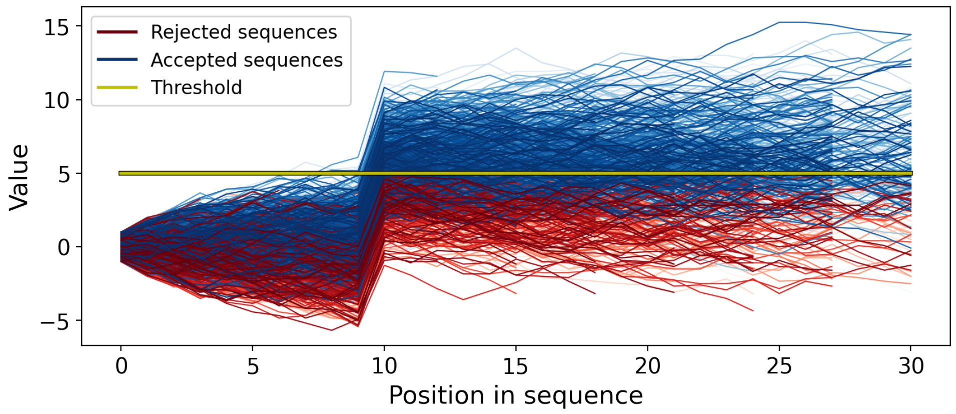


Figure 4.7: Sequence dataset using an alphabet of size 25. After 10 steps the "value" of each sequence is affected by external influence and increased by 5.

demonstrate the change in complexity associated with an increased alphabet size. Figure 4.7 displays the dataset affected by external influences. Here the value of the sequences increases with 5 after 10 steps. This particular figure displays the dataset created using an alphabet size of 25. All datasets consist of sequences of lengths [1, 4, 7, ..., 31] with 2000 random samples for each length[2]. Training and test data is 85% and 15% of the dataset, and 10 000 random sampled sequences are used for the long test data. The percentage of positive data are ∼35% and ∼49% for the datasets with and without external influences respectively. The LSTM models uses 2 layers with 64 nodes in each layer. The training uses a batch size of 64 and a dropout of 0.3. All models are trained until reaching an F1 score of 1 on the training data. Rule extraction uses a refinement depth of 10, max extraction time of 900 seconds, and starting example lengths of 1. The value threshold used to label sequences is 2 and 5 for each dataset. Results are averaged over 3 runs.

No External Influences

A small extracted DFA on this dataset using an alphabet size of 10 is illustrated in figure 4.8. The DFA has learned that after reaching an accepting state (state 7) all other transitions should lead to the same accepted state as well. Table 4.6 gives the F1 scores for the LSTM models and the extracted DFA on training data and test data. All LSTM models score high on the evaluation datasets. In contrast, the extracted DFA show varying results on the evaluation data, with lower scores than their respective LSTM models. Only for the smallest alphabet of size 3 does the algorithm converge

[2]Short sequences with few total symbol combinations (less than 2000 in this case), use all possible combinations instead.

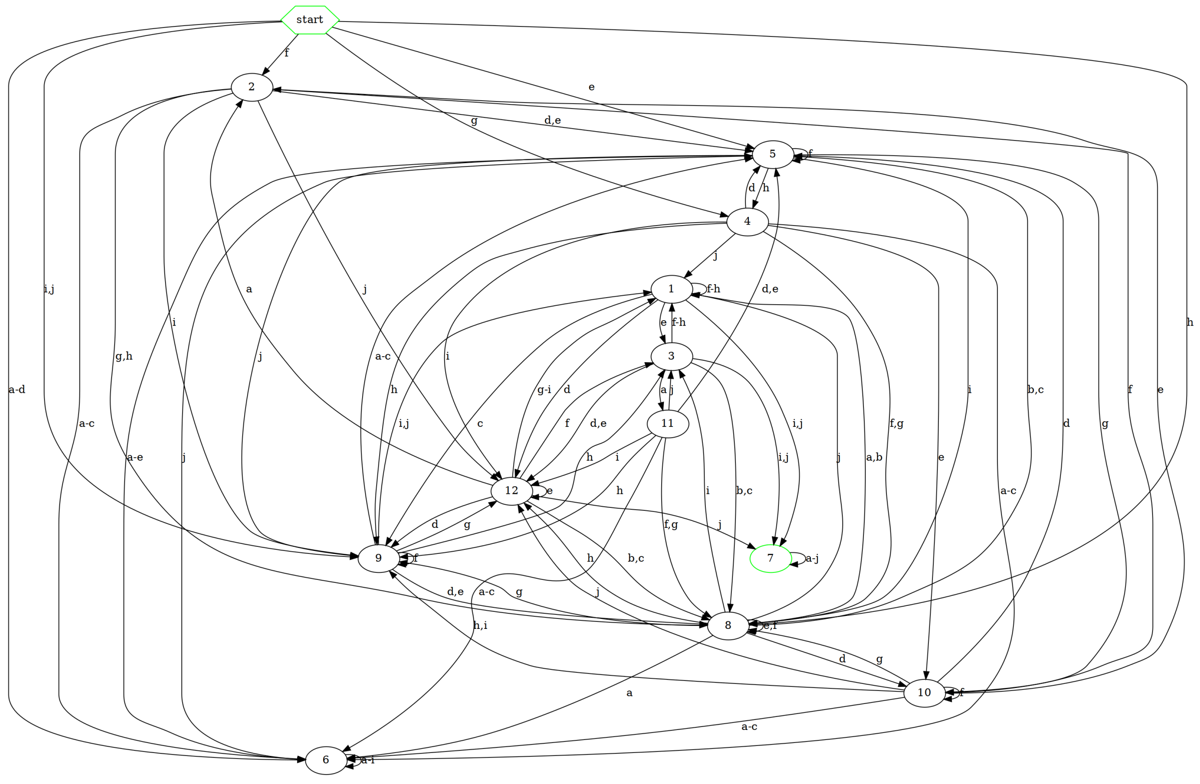


Figure 4.8: Extracted DFA using using alphabet size 10 on the action value dataset. 13 nodes.

Model	Alphabet size	F1 score train data	F1 score test data	F1 score long test data	DFA size	Converged
LSTM4	3	1.0	1.0	0.997		
DFA4	3	0.894	0.895	0.843	6	Yes
LSTM5	10	1.0	0.982	0.972		
DFA5	10	0.726	0.724	0.803	149.3	No
LSTM6	25	1.0	0.961	0.972		
DFA6	25	0.377	0.382	0.335	91.3	No

Table 4.6: Performance of models and extracted DFA, no external influence.

toward a solution before the time limit. All 3 runs return the same DFA. Notably, these DFA are the best-performing extractions on the evaluation datasets, indicating a better extraction quality on simpler datasets. Furthermore, we observe that the extraction using an alphabet of size 25 has a considerably lower average F1 score compared to the other extractions. A reason for this might be that with the increasing complexity of the problem through the use of a larger alphabet, comes an increasing difficulty of extracting accurate DFA. With this in mind, the extracted DFA of the dataset with alphabet of size 10, achieves an impressive F1 score.

External Influence

Here we assess how external influences independent of the input affect rule extraction. The datasets are similar to the previous test case, with the additional modification of increasing sequence values by 5 after 10 steps and using a higher threshold. Most sequences start out positive as the threshold is set relatively high compared to the initial values.

Model	Alphabet size	F1 score train data	F1 score test data	F1 score long test data	DFA size	Converged
LSTM7	3	1.0	0.999	0.954		
DFA7	3	0.995	0.995	0.953	933	No
LSTM8	10	1.0	0.986	0.989		
DFA8	10	0.100	0.101	0.229	77.3	No
LSTM9	25	1.0	0.981	0.976		
DFA9	25	0.013	0.010	0.017	45	No

Table 4.7: Performance of models and extracted DFA, with external influence.

Table 4.7 gives the scores and information of the extracted DFA and their LSTM models on the sequence value dataset with external influences. Firstly, we observe a substantial difference in the F1 scores of the DFA compared to the previous test case for alphabet sizes 10 and 25. For these alphabet sizes the extracted DFA score is very low on all evaluation data, indicating a more difficult dataset for rule extraction. Despite this, the DFA extracted on the dataset with alphabet size 3 scores very high, again showcasing the difference in extraction quality with regard to the complexity of the problem. In this case, DFA7 does not converge to a DFA within the time limit. In addition, the extracted DFA has noticeably lower node counts for alphabet sizes 10 and 25 compared to the previous test case. Possibly due to building a large abstraction and spending much time exploring and refining it. Resulting in that for the total given extraction time, the algorithm is only capable of extracting a small DFA early in the extraction processes, and for the remainder of the extraction time, is unable to build a better DFA.

By employing datasets with and without external influences, we get an indication of the influence of problem complexity as well as alphabet complexity, on the quality of rule extractions. The increased complexity through the sudden change in sequence values for this type of dataset negatively affects the quality of the extracted DFA. Additionally, for both test cases, increasing alphabet size reduces the overall quality of the rule extraction. Furthermore, with regard to the smallest alphabet datasets for both test cases, we see a lower F1 score for the case with no external influences, despite being a simpler problem. In this case, the rule extraction converged to a DFA of size 6. In contrast, for the case with external influences, the algorithm did not converge and reached a higher F1 score. This indicates that the DFA might have terminated too early and would most likely have found a better DFA if not for the premature termination.

4.6 Test Case Results for the Well-Drilling Dataset

In the previous section, we observed a change in the quality of the rule extraction based on increased complexity from both increased alphabet size and the addition of external influences. For the final test case, we examine the performance of the rule extraction approach on a real-life inspired physical process, emulating a well-drilling process. This dataset has similar traits to the sequence value datasets. The well-drilling dataset is generated through a hybrid approach of utilizing the well-drilling simulator OpenLab and the SINDy algorithm.

4.6.1 Sparse Identification of Nonlinear Dynamical Systems

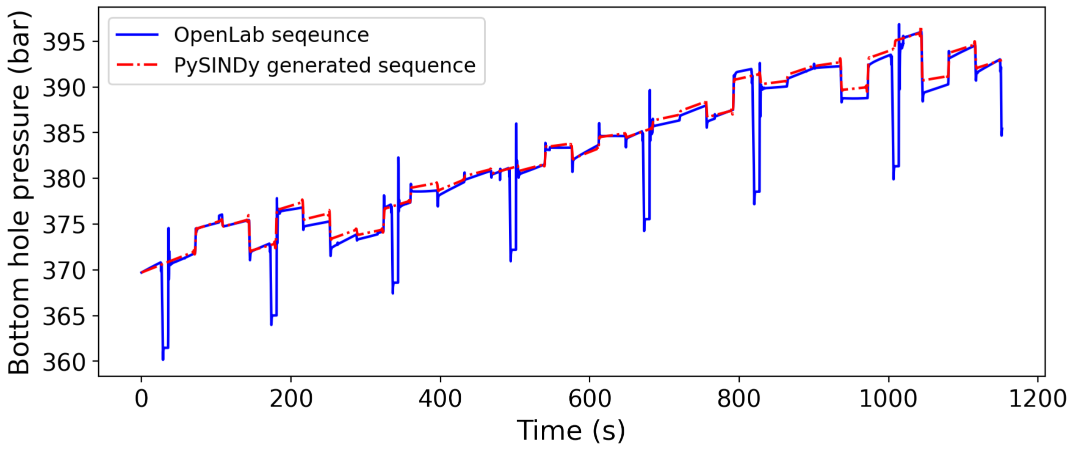
In section 3.2.3, we address the impracticality of generating thousands of sequences directly using the OpenLab simulator. For this reason, we employ the SINDy algorithm to obtain an expression of the very specific well-drilling system we utilize for our experiments, allowing us to generate data for training and testing faster and more efficiently than directly through OpenLab. Prior to conducting rule extraction on models trained on the well drilling dataset, we assess the quality of the synthetic data generated using PySINDy.

The system identified by PySINDy is expressed as follows:

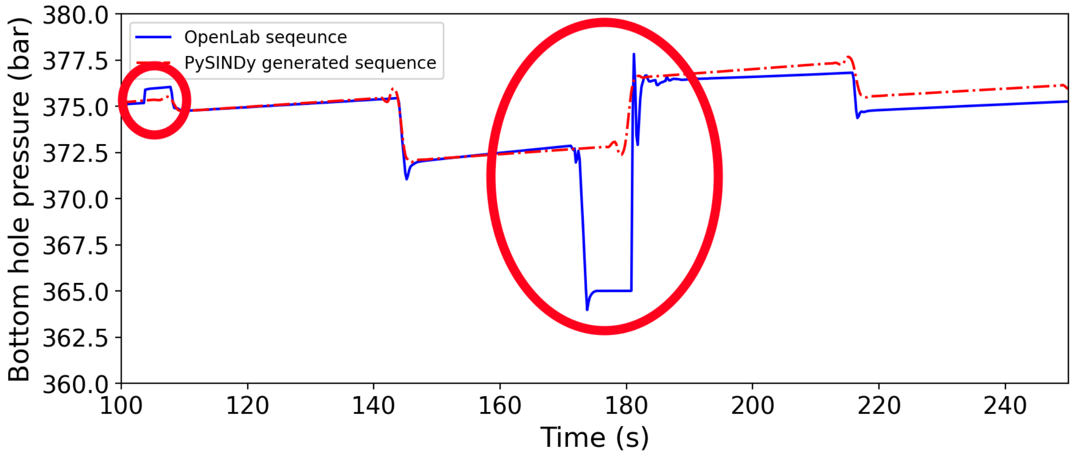
$$(x)' = 0.11606 - 0.00029x + 0.00023b + 0.00301\frac{d^1a}{dt^1} + 0.00834\frac{d^1b}{dt^1} - 0.00182\frac{d^2a}{dt^2} - 0.00793\frac{d^2b}{dt^2} + 0.00057\frac{d^3a}{dt^3} + 0.00202\frac{d^3b}{dt^3} - 0.00137\frac{d^4b}{dt^4} + 0.00011\frac{d^5a}{dt^5} + 0.00051\frac{d^5b}{dt^5} \quad (4.1)$$

Here x is the BHP, a and b are the flow rate and RPM parameters respectively. The differential equation is solved by starting from an initial pressure value x_0.

Using 4.1 we compute the BHP sequences using inputs of flow rate, RPM, and their respective derivatives. Figure 4.9 compares a synthetic PySINDy sequence and an OpenLab sequence generated from the same input parameters obtained from the test data. The equation found by PySINDy has computed a synthetic simulation very similar to the OpenLab sequence, and has managed to capture the defining characteristics of the data. Nevertheless, we observe that the PySINDy sequence is unable to capture the drilling auto-connection sections characterized by the steep drop in pressure. It also struggles with the more complex features such as the artifacts highlighted in 4.9b. Despite these shortcomings, our experiments mainly focus on a given upper threshold constraint, meaning that pressure drops from the auto-connection sections are not of importance. Furthermore, we sample at a maximum of 31 values from the sequences, using the halfway value between the setpoints as seen back in figure 3.7, reducing the importance of recre-



(a) Full sequence.



(b) A section of the sequence.

Figure 4.9: SINDy generated BHP sequence compared to OpenLab simulated BHP sequence.

ating the more complex parts of the simulation. The data generated using equation 4.1 found by SINDy, give a good approximation of the OpenLab simulation data for these test cases.

4.6.2 Experiment Configuration

100 simulations from OpenLab are used to train a PySINDy model that in turn generates 1000 sequences for each of the lengths [1, 4, 7, ..., 31], using equation 4.1. Sequences are generated using 5 flow rate values [700, 1200, 1700, 2200, 2700]l/min and 5 RPM values [20, 45, 70, 95, 120] rpm, resulting in an alphabet size of 25. Sequence labeling utilizes different values of k. k is the number of values in a row above the threshold that results in an acceptance of the sequence. In this case, the threshold is a sequence of values, that aim to imitate some pressure limitations that could be present in a drilling scenario. The resulting datasets have positive data distributions of 50.1%, 25.2% and, 14.8% for k=1, k=2 and k=3 respectively, resulting in particularly imbalanced datasets for k=2 and k=3. Figure 4.10 and 4.11 show the SINDy generated BHP sequences before and after sampling for k=1. The LSTM models use 2 layers each with 64 nodes and a dropout of 30%. The models were trained until an F1 score of 1.0 was reached. Rule extraction uses a refinement depth of 10, starting example length 1, and a max extraction time of 900 seconds. Results are averaged over 3 runs.

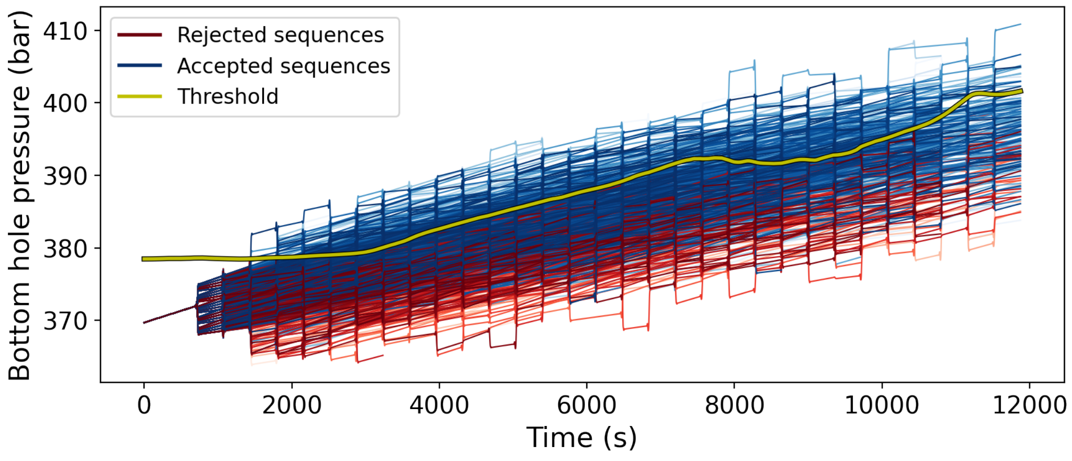


Figure 4.10: Full SINDy generated BHP sequences based on simulated data from OpenLab. k=1.

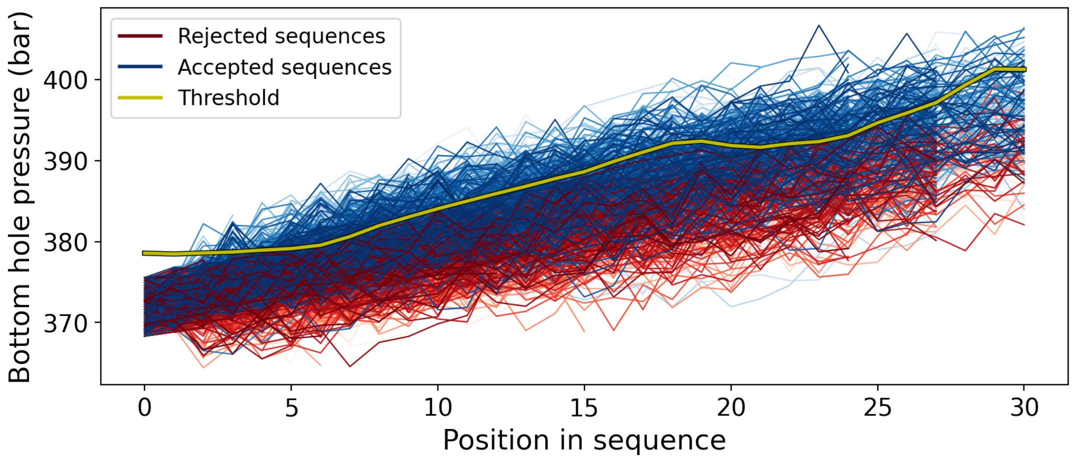


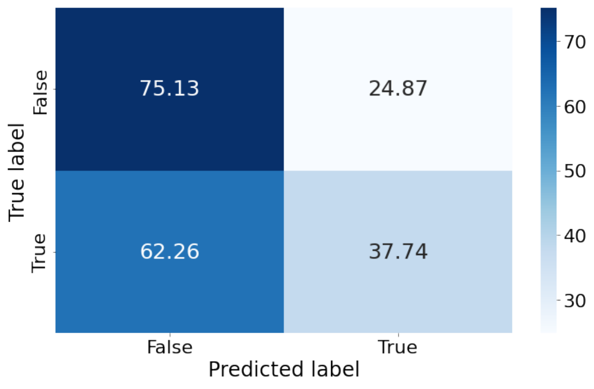
Figure 4.11: Sampled SINDy generated BHP sequences based on simulated data from OpenLab. k=1

Model	k	F1 score train data	F1 score test data	DFA size	Converged
LSTM10	1	1.0	0.753		
DFA10	1	0.316	0.292	468.3	No
LSTM11	2	1.0	0.510		
DFA11	2	0.243	0.266	523.67	No
LSTM12	3	1.0	0.365		
DFA12	3	0.082	0.081	119.67	No

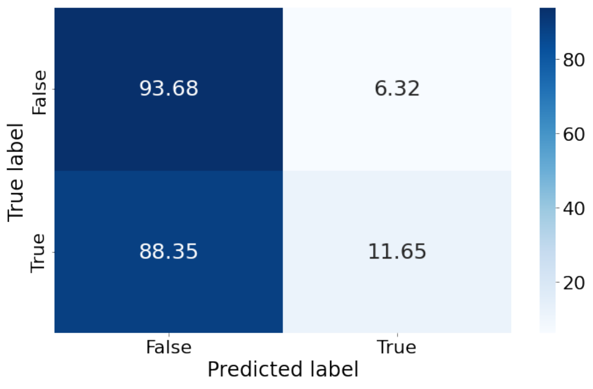
Table 4.8: Performance of models and extracted DFA, on the well-drilling dataset.

Results

Table 4.8 show the F1 scores of the trained LSTM models and their extracted DFA on training and evaluation data. We see that for higher k comes lower scores. For k=1 we observe a low F1 score of 0.316 for the extracted DFA, compared to the LSTM network with a score of 1.0 on the training data. However, we see that the LSTM model has not generalized very well on the datasets, and scores only 0.753 on the test data compared to its perfect score on the training data. This same pattern can be seen for k=2 and k=3 as well, only with much lower F1 scores for the DFA and LSTM models on the test data. For $k = 3$ the LSTM model scores 1.0 and 0.365 on the training and test data respectively, indicating that the model is overfitted. The confusion matrices for extracted DFA with k=1 and k=3, evaluated on the test data are compared in figure 4.12. The confusion matrices show the prediction percentage of each class label. Notably, we see that for k=3 the extracted DFA ends up predicting almost all sequences as false. Consequently, the DFA receives especially low F1 scores, as the metric put emphasis on correctly predicting sequences exceeding the threshold.



(a) Confusion matrix for k=1



(b) Confusion matrix for k=3

Figure 4.12: Percentage prediction distribution for each classification label.

Reasons for the poor performance of higher k can possibly be attributed to an increased complexity of the datasets, similarly to the observations made from the sequence value datasets. Nevertheless, we observe a notable difference between the experiments. With the introduction of external influences in the sequence value dataset, we experience a steep drop in the quality of the extracted DFA for alphabets of sizes 10 and 25. The sequence value dataset achieves an F1 score of 0.1 on the training data using an alphabet of size 10. In comparison, the well-drilling dataset for k=1 and k=2 scores 0.316 and 0.243 on the training data, despite using a large alphabet of size 25 for all k. It is possible that with the introduction of external influences, the sequence value dataset becomes a particularly difficult problem for the rule extraction algorithm. This is most likely due to the sudden change in sequence values after 10 steps. In contrast, the BHP and the threshold of the well-drilling dataset increases more evenly, possibly resulting in a better-suited problem for rule extraction.

4.7 Size and Inference Time Comparisons

Depending on the scenario, the LSTM model and the extracted DFA perform the same action of making predictions based on a given input. However, in some cases, it can be beneficial to employ smaller and faster models. Some examples of this can be in production to reduce cost, in small applications with memory restriction, and in applications requiring fast inference time for real-time processing. We compare memory usage and inference time of extracted DFA and their respective LSTM model for two of the previous datasets:

- **Dataset1:** Sine wave dataset with alphabet size 5. Train data 26812 samples.
- **Dataset2:** Well drilling dataset for $k = 1$ with alphabet size 25. Train data 10650 samples.

Inference time is measured using a batch size of 1024 for the LSTM model. In this case, predictions are faster when using larger batches, however at the cost of additional memory needed to store the batches temporarily. Additionally, we note that the inference time for LSTM models and DFA is dependent on the available hardware, as better hardware will give faster predictions. Predictions are done using the GPU for the LSTM networks and the Central Processing Unit (CPU) for the DFA. These were the fastest options for each network. Inference time and accuracy are averaged over 5 runs using the training data.

Model	Dataset	Complexity	F1 score train data	Size(kb)	Total inference time(s)
LSTM13	1	2 layer 64 nodes	1.0	3985	1.69
DFA13	1	13 nodes	1.0	2	0.047
LSTM14	2	2 layer 64 nodes	1.0	3993	0.689
DFA14	2	465 nodes	0.499	75	0.030

Table 4.9: Table comparing F1 score, size, and inference time between extracted DFA and LSTM models.

The comparisons in table 4.9 demonstrates that the extracted DFA uses considerably less memory than their LSTM counterparts and has a much lower inference time. All models for this comparison are stored on disk using the Python object serialization module `pickle` (docs.python.org). Inference time is measured using Python's `time()` method from the `time` module. Note, we measure total inference time over the full training dataset. Comparing LSTM13 trained on dataset 1 and its extracted DFA, we observe that both models predict the same perfect F1 score. However, LSTM13 using 2 layers and 64 nodes use 3.99MB of memory, in contrast to the extracted DFA which is about 2000 times smaller and only uses 0.002MB. In addition, the DFA has an inference time that is close to 36 times faster. Regarding LSTM14 and DFA14, we see the same pattern with the DFA being smaller and faster. However, because of its larger node count, DFA2 takes up more memory than DFA1 and is about only 53 times smaller in size compared to LSTM2. Notably, DFA14, despite its fast inference time and low memory usage, has a very low F1 score compared to LSTM14. In this regard, the low scores raise questions about the practicality of DFA extracted from models trained on datasets of this type.

Chapter 5

Discussion and Analysis of Results

The following sections analyze and give interpretations of the results and experiments provided in chapter 4. Furthermore, we discuss the datasets employed in this study and address limitations and potential benefits of rule extraction on these types of datasets.

5.1 Sparse Identification of Nonlinear Dynamical Systems

In this thesis, we have utilized the SINDy algorithm on simulated data from OpenLab to facilitate a faster and more efficient process of generating synthetic well-drilling data. Originally, all data for the well-drilling dataset was planned to come from the OpenLab simulator. However, it became clear that generating the required amount of data needed for training LSTM models solely from OpenLab was impractical, necessitating the adoption of another data generation approach. Due to our simulations primarily being dominated by linear features, the SINDy algorithm was deemed as an appropriate choice for approximating the simulation system through basic terms.

For our SINDy implementation, we ended up using only two control parameters, flow rate and RPM, to predict BHP. To achieve a more detailed representation of the simulations, we could experiment with using more control parameters, such as WOB or ROP. Another approach could be to express more of the system at the same time, for instance, drilling depth and/or well temperature together with BHP, in order to get a more holistic understanding of the system. However, we conclude that in our case, it was enough to use two control parameters and only predict the BHP to obtain satisfactory synthetic data, capable of capturing the most distinct features of the simulation. The control parameter values were randomly selected from intervals specifically chosen to achieve significant variations in the BHP. To express more specific drilling scenarios, smaller and more specific parameter value intervals could be used. In practice we ended up with an approximation of the OpenLab simulations, improving speed and efficiency at the cost of precision. The most notable limitation of the SINDy generated data was its inability to express the pressure drop resulting from auto-connections. However, this aspect was not important with regards to our experiments. Overall, in this thesis, the SINDy algorithm proved to be a highly capable data generation tool, facilitating evaluations of the rule extraction algorithm on synthetic real-world processes.

Note, it is difficult to fully describe such a complex and intricate well-drilling system accurately. A full well-drilling simulation computes much more and manages many more parameters than the approximated model found in this thesis, such as temperature and well fluid density. Small adjustments to configurations and parameters can result in significant changes in the simulations. For these reasons, we do not expect that the system found by SINDy in this thesis can directly be applied to other well-drilling configurations with the same success. For other scenarios, new SINDy models should be trained.

5.2 Datasets

The datasets employed in this thesis explore and experiment with the limitations of the rule extraction algorithm for problems of varying complexity. Here we give some remarks on aspects of the datasets that should be considered.

Perfect Data

We highlight that all datasets have been created synthetically without the presence of noise or other unwanted influences, which can often be present in more realistic data. For example, in real-world well-drilling processes, it is common for collected data to suffer from noise in the form of missing and faulty sensor data, processing errors, and incorrect measurements. For this thesis, the intention was to experiment using noise-free data, as the algorithm is sensitive to the presence of noise, as stated by (Weiss et al., 2018). Our aim was to examine how complexities in datasets could be represented through extraction of DFA, rather than finding ways to handle noise. Despite this, we observe another type of noise affecting the rule extractions of LSTM models. Models that fail to generalize on the dataset are susceptible to adversarial examples, and as a result, can result in extraction of wrong DFA. For the sine wave dataset, we see the effect of this even when the model manages to predict all sequences of the dataset correctly.

Data Imbalance

For all datasets, the binary labels of the samples are defined by us. In most experiments, we have decided on a labeling threshold with the aim to prevent highly unbalanced datasets. Deciding the class balance of datasets might not be possible in real-world scenarios. Despite this, the majority of datasets in this thesis still end up with imbalanced classification distributions. The sequence value dataset and the well-drilling dataset have both a prevalence of negative data comprising about 50% - 70% of the overall distribution. For the well-drilling dataset, we observe a much larger class imbalance with the introduction of larger k to the drilling dataset. These datasets produce very low F1 scores and are unable to predict almost any of the positive sequences, possibly affected by the large class imbalance. Exploring the effect of unbalanced datasets on the rule extraction approach could serve as an interesting avenue for future research.

Data Amount

Notably, we observe that the LSTM models, for some experiments, are unable to generalize on the dataset. This is particularly evident in the well-drilling dataset (see table 4.8). Despite scoring perfect on training data, the models scores poorly on the test data. Extracted DFA from these LSTM models have low scores. Additionally, recall that LSTM models trained on the sine wave dataset were shown to be susceptible to adversarial examples, allowing for inaccurate predictions to be inherited by extracted DFA. These considerations suggest that better generalized models can improve on the quality of rule extraction. Nonetheless, it can be difficult to know if a satisfactory generalization is reached. As experienced with the sine wave dataset, predicting all sequences correctly still allows for incorrectly accepting wrong inputs. The notion of test data scores being an untrustworthy evaluation measure is also highlighted by Weiss et al. in their work. Improved model generalization can be achieved by increasing the amount of training data and using larger models. Nevertheless, in this thesis, due to available computational resources, constraints are imposed on the amount of generated data and the size of the

LSTM models. These constraints have in particular influenced the size of the well-drilling dataset. Although leveraging the PySINDy equation facilitates faster and more efficient computations, computing 3 hours of drilling data is still a resource-intensive task. In this regard, to speed up computations we could reduce the complexity of the SINDy model or alternatively, downsample the data before training the SINDy model. However, this could result in a worse approximation of the simulation data.

5.3 Evaluation Metric

In this thesis, the F1 score has been selected as the main evaluation metric. The main advantage of using the F1 score is the ability to focus on the correct prediction of a specific class during experimentation. In this work, we experiment with theoretical safety-critical predictions, where the consequences of wrongly predicting sequences exceeding a given constraint can be dangerous. It is important to note that F1 scores do not take into account true negatives, as can be seen in equation 2.23. Possibly resulting in high F1 scores for cases where models predict all sequences as true for datasets with class distribution dominated by positive classes. The opposite of this can be observed in the well-drilling experiments, where extracted DFA evaluated on test data receives low F1 scores, despite correctly predicting almost all negatively labeled sequences. This is illustrated by the confusion matrices 4.12a and 4.12b. We suggest for the evaluation of datasets affected by class imbalance, where both classes are equally important, to use an evaluation metric that considers the correct predictions of all classes, such as Matthew's correlation coefficient (MCC). MCC only returns high scores if both classes are predicted accurately (Grandini et al., 2020).

5.4 Extracted DFA Discussion

Performance Evaluation

Throughout our experiments, we observe that the rule extraction algorithm is able to find simpler and smaller representations of LSTM networks using DFA. In particular, we see that DFA extracted using small alphabets produces the best results. Even the complex sequence value dataset with external influences reaches a high average F1 score of 0.995 on the test data for an alphabet of size 3. However, we also observe that smaller alphabets are more prone to early terminations, which for our experiments with the sequence value dataset with no external influences, results in lower-quality DFA extractions. We notice that the increasing complexity of the datasets results in difficult rule extractions, and consequently worse predictive DFA. Overall, it is possible that the chosen datasets in this work have been too ambitious with regard to complexity. The low scores using alphabets of size 25 on the sequence value datasets and well-drilling datasets suggest that the rule extraction approach is impractical for such datasets. Exploring real-world processes that are better suited for this rule extraction approach is an interesting avenue for future work.

Drawbacks

We highlight some drawbacks of extracting DFA in this manner. First, discretizing data to be represented using small alphabet sizes may result in bad representations of the original data. As stated, our best results come from using small alphabet sizes. Finding cases where it is practical to represent a problem or dataset using only, for instance, 3 or 4 values might not be trivial. So despite scoring high in the evaluations, the practicality of small alphabet DFA varies across applications. Secondly, it is worth noting that in general, DFA are less capable than LSTM models. This is a consequence of the much simpler structure of the DFA compared to the non-linear LSTM models. Although some of the extracted DFA in the sine wave dataset demonstrated capabilities of giving more accurate predictions than their LSTM models. For other cases in this work, the extracted DFA was not able to make as good predictions as the LSTM models. For such cases, it must be decided if the loss of prediction accuracy is worth the size reduction, increased inference speed, and the improved model comprehension. Overall, we suggest that DFA extraction can be successfully employed on LSTM models trained on some specific continuous datasets that are inherently simple and lend themself to representation through small alphabets.

5.5 Future Work

In this thesis, we have looked at the practicality of applying rule extraction on complex problems imitating real-world physical processes. We present some interesting directions for future development and research on topics brought up in this thesis.

Although this approach of rule extraction might not have been the perfect fit for the well-drilling problem, given its complexity and alphabet size. It is still of interest to find and experiment with applications for other real-world datasets. Especially, finding real-world scenarios that fit better to the strengths of the rule extraction algorithm examined in this work. In particular problems that are suited for discretization using small alphabets and are not overly complex. Furthermore, using different approaches for rule extraction on the same or similar experiments performed in this thesis, can be interesting. Such as the compositional approaches described by Jacobsson (2005). Some of these rule extraction approaches, provide better scalability at the cost of higher extraction times. Another natural progression from this thesis is to conduct the same experiments with more computational resources. Allowing for training larger models on larger datasets, and for conducting faster DFA extractions. This could help with training better generalized models and could provide more insight into how extracted DFA size influences predictions. Considering the small size and high inference speed of the extracted DFA compared to the LSTM models, it could be interesting to experiment with ways to combine multiple small accurate DFA to solve more complex tasks. For example, in multi-class prediction problems, multiple DFA could be extracted from separate LSTM models trained on binary predicting each of the given classes. The set of DFA could be combined into a single system used for solving non-binary prediction problems, with the benefit of being faster and using less memory than a LSTM model trained to solve the same problem.

Chapter 6

Conclusion

In this thesis, we have undertaken a comprehensive examination and evaluation of the feasibility of applying a rule extraction approach, developed by Weiss et al. (2018), on LSTM models trained on increasingly complex data aimed at simulating physical processes. Multiple DFA has been extracted using various datasets comprising of sequences of discretized continuous values. Specifically, we have utilized self-generated datasets, encompassing a sine wave dataset and a sequence value dataset, as well as a safety-critical well-drilling pressure scenario. Data for the well-drilling scenario have been obtained by employing the SINDy algorithm on simulation data from the high-fidelity well-drilling simulator OpenLab. For our particular problem and simulation configuration, SINDy proved to be a very capable tool for data generation, allowing for faster and more efficient data generation.

Comparisons between extracted DFA and the LSTM models used for extraction trained on the considered datasets, show that in most cases the DFA are unable to be as accurate as the LSTM models. In particular, we see a clear difference in performance with increasing complexity of the datasets, and with an increasing alphabet size used to discretize the data. Smaller alphabet sizes produce the highest quality DFA and increasing complexity results in lower quality DFA. For the sine wave dataset, the rule extraction approach is capable of finding minimal DFA that are even more robust and accurate than their respective LSTM models. In this case, the extracted DFA represents all states and transitions that define a sine wave and does not accept difficult to predict adversarial inputs, in contrast to the LSTM models. Nevertheless, experiments on more complex problems, such as the sequence value dataset and the well-drilling dataset, results in extracted DFA unable to make as good predictions as the LSTM models. Furthermore, we have examined the difference in memory usage and inference time for LSTM models and extracted DFA. Noting that the extracted DFA are both significantly smaller in size and exhibit vastly lower inference times compared to their respective LSTM models.

Overall, for the experiments conducted in this thesis, we observe that the rule extraction algorithm can successfully be applied to datasets consisting of continuous values by both discretizing the values using few symbols and using simple datasets. In contrast, for more complex problems we observe extractions of inaccurate DFA when compared to the predictive quality of their LSTM models. Thus, based on our findings, we suggest that the rule extraction algorithm should be applied to certain problems that are able to utilize the strengths of the algorithm, in order to achieve the best results. These applications should be simple and have datasets capable of being represented through discretization using few symbols.

Bibliography

Oludare Isaac Abiodun, Aman Jantan, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(87)90052-6. URL `https://www.sciencedirect.com/science/article/pii/0890540187900526`.

Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. *Electronic Notes in Theoretical Computer Science*, 118:3–18, 2005. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2004.12.015. URL `https://www.sciencedirect.com/science/article/pii/S1571066104053216`. Proceedings of the International Workshop on Software Verification and Validation (SVV 2003).

Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016. doi: 10.1073/pnas.1517384113. URL `https://www.pnas.org/doi/abs/10.1073/pnas.1517384113`.

John Carroll and Darrell Long. *Theory of Finite Automata with an Introduction to Formal Languages*, pages 23–65. Prentice-Hall, Inc., USA, 1989. ISBN 0139137084.

Brian M. de Silva, Kathleen Champion, Markus Quade, Jean-Christophe Loiseau, J. Nathan Kutz, and Steven L. Brunton. Pysindy: A python package for the sparse identification of nonlinear dynamical systems from data. *Journal of Open Source Software*, 5(49):2104, 2020. doi: 10.21105/joss.02104. URL `https://doi.org/10.21105/joss.02104`.

docs.python.org. Pickle — python object serialization. URL `https://docs.python.org/3/library/pickle.html`. [Accessed May 28, 2023].

Timothy Dozat. Incorporating Nesterov Momentum into Adam. In *Proceedings of the 4th International Conference on Learning Representations*, pages 1–4, 2016.

Robert Ewald, Jan Einar Gravdal, Dan Sui, and Roman Shor. Web enabled high fidelity drilling computer model with user-friendly interface for education, research and innovation. volume 153, pages 162–168. Linköping University Electronic Press, 11 2018. doi: 10.3384/ecp18153162.

LSTAR Extraction. Lstar extraction, 2022. URL `https://github.com/tech-srl/lstar_extraction`. [Accessed 17 May 2023].

John Cristian Borges Gamboa. Deep learning for time-series analysis. *ArXiv*, abs/1701.01887, 2017.

Salvador García, Julián Luengo, José Antonio Sáez, Victoria López, and Francisco Herrera. A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Transactions on Knowledge and Data Engineering*, 25:734–750, 2013. ISSN 10414347. doi: 10.1109/TKDE.2012.35.

Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: an overview. *ArXiv*, abs/2008.05756, 2020.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*, pages 1–55. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321455363.

Henrik Jacobsson. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Computation*, 17(6):1223–1263, 2005. doi: 10.1162/0899766053630350.

Justin M. Johnson and Taghi M. Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6, 12 2019. ISSN 21961115. doi: 10.1186/s40537-019-0192-5.

Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of Atmospheric Sciences*, 20(2):130 – 141, 1963. doi: https://doi.org/10.1175/1520-0469(1963)020⟨0130:DNF⟩2.0.CO;2. URL `https://journals.ametsoc.org/view/journals/atsc/20/2/1520-0469_1963_020_0130_dnf_2_0_co_2.xml`.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rJzIBfZAb`.

David Martens, Johan Huysmans, Rudy Setiono, Jan Vanthienen, and Bart Baesens. *Rule Extraction from Support Vector Machines: An Overview of Issues and Application in Credit Scoring*, pages 33–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-75390-2. doi: 10.1007/978-3-540-75390-2_2. URL `https://doi.org/10.1007/978-3-540-75390-2_2`.

Microsoft. Windows subsystem for linux, 2023. URL `https://learn.microsoft.com/en-us/windows/wsl/`. [Accessed May 22, 2023].

Marvin L. Minsky. *Computation: Finite and Infinite Machines*, pages 32–58. Prentice-Hall, Inc., USA, 1967. ISBN 0131655639.

W. James Murdoch and Arthur Szlam. Automatic rule extraction from long short term memory networks. *arXiv preprint arXiv:1702.02540*, 2017.

Christian W. Omlin and C. Lee Giles. *Symbolic Knowledge Representation in Recurrent Neural Networks: Insights from Theoretical Models of Computation*, page 63–116. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262032740.

OpenLab. Webclient user guide, 2022. URL `https://openlab.app/user-guide/`. [Accessed May 22, 2023].

OpenLab. Openlab, 2023a. URL `https://openlab.app/`. [Accessed May 22, 2023].

OpenLab. Openlab, 2023b. URL `https://openlab.app/about/`. [Accessed May 22, 2023].

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.

Marius-Constantin Popescu, Valentina Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8, 07 2009.

PyTorch. Welcome to pytorch tutorials, a. URL `https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html`. [Accessed May 28, 2023].

PyTorch. Pytorch examples, b. URL `https://github.com/pytorch/examples/blob/main/word_language_model/model.py`. [Accessed May 28, 2023].

Andrinandrasana David Rasamoelina, Fouzia Adjailia, and Peter Sinčák. A review of activation function for artificial neural network. In *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 281–286, 2020. doi: 10.1109/SAMI48414.2020.9108717.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

Nejm Saadallah, Jan Einar Gravdal, Robert Ewald, Sonja Moi, Adrian Ambrus, Benoit Daireaux, Stian Sivertsen, Kristian Hellang, Roman Shor, Dan Sui, Stefan Ioan Sandor, Marek Chojnacki, and Jacob Odgaard. OpenLab: Design and Applications of a Modern Drilling Digitalization Infrastructure. volume Day 1 Tue, May 14, 2019 of *SPE Norway Subsurface Conference*, 05 2019. doi: 10.2118/195629-MS. URL `https://doi.org/10.2118/195629-MS`. D011S002R002.

Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *ArXiv*, abs/1912.05911, 2019.

STLSQ. Pysindy 0.13.2 documentation: pysindy.optimizers, 2023. URL `https://pysindy.readthedocs.io/en/latest/api/pysindy.optimizers.html#module-pysindy.optimizers.stlsq`. [Accessed May 17, 2023].

Qinglong Wang, Kaixuan Zhang, Alexander Ororbia, Xinyu Xing, Xue Liu, and C. Lee Giles. A comparative study of rule extraction for recurrent neural networks. *arXiv: Learning*, 2018a.

Qinglong Wang, Kaixuan Zhang, Alexander G. Ororbia II, Xinyu Xing, Xue Liu, and C. Lee Giles. An Empirical Evaluation of Rule Extraction from Recurrent Neural Networks. *Neural Computation*, 30(9):2568–2591, 09 2018b. ISSN 0899-7667. doi: 10.1162/neco_a_01111. URL `https://doi.org/10.1162/neco_a_01111`.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, Stockholmsmässan, Stockholm, Sweden, 10–15 Jul 2018. PMLR.

Yan Xiao, Ivan Beschastnikh, David S. Rosenblum, Changsheng Sun, Sebastian Elbaum, Yun Lin, and Jin Song Dong. Self-checking deep neural networks in deployment. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 372–384, 2021. doi: 10.1109/ICSE43902.2021.00044.

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021a.

Xiyue Zhang, Xiaoning Du, Xiaofei Xie, Lei Ma, Yang Liu, and Meng Sun. Decision-guided weighted automata extraction from recurrent neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13):11699–11707, May 2021b. doi: 10.1609/aaai.v35i13.17391. URL `https://ojs.aaai.org/index.php/AAAI/article/view/17391`.

Appendix A

LSTM Model Code

We include the code for the LSTM model mentioned in section 3.3 incorporating the L* rule extraction methods described in their code repository Extraction (2022). The implementation inherits from the `torch.nn.Module` class, the base class for all neural network models in PyTorch. The model design is based on examples found in PyTorch's code base (PyTorch, a) and (PyTorch, b).

L* rule extraction methods follow the implementation description by Weiss et al. in their publicly available code repository Extraction (2022). As well as adding a list variable of the symbols of the alphabet.

```python
import torch
from torch import nn
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence



class LSTM_model1(nn.Module):
    """
    LSTM recurrent network cell.
    Implementing L* rule extraction API methods from:

    Paper:
    https://arxiv.org/pdf/1711.09576.pdf

    Code repository:
    https://github.com/tech-srl/lstar_extraction
    """
    def __init__(self,
                 hidden_size,
                 out_size,
                 n_layers,
                 embedding_dim,
                 symbol_dict,
                 device,
                 dropout=0.3):

        """
        Args:
            hidden_size: (int), Number of nodes of the hidden layer(s) of the
                         LSTM cell.
            out_size: (int), Number of output features of the LSTM cell.
            n_layers: (int), Number of stacked LSTM layers
            embedding_dim: (int), Size of embedding vectors for each symbol in the
                            input sequence.
            symbol_dict: (dict), Dictionary mapping symbols to respective indecies
            device: (torch.device), Device used for training and running (e.g.
                    "cuda", "cpu"). Used for evaluation to know what device
                    tensors should be put on.
            dropout: float, Percentage of nodes to temporarily drop.
        """

        super().__init__()
        self.device = device
        self.symbol_dict = symbol_dict
        self.n_layers  = n_layers
        self.hidden_size = hidden_size
        self.alphabet = list(symbol_dict.keys())

        num_embeddings = len(symbol_dict)+1 # Account for the empty word

        self.embedding = nn.Embedding(num_embeddings, embedding_dim,
                                      device=self.device, dtype=torch.double)
        self.lstm      = nn.LSTM(embedding_dim, self.hidden_size, self.n_layers,
                                 batch_first=True, dropout=dropout)
        self.fc        = nn.Linear(self.hidden_size, self.hidden_size)
        self.relu      = nn.ReLU()
        self.dropout   = nn.Dropout(dropout)
        self.output    = nn.Linear(self.hidden_size, out_size)

    def process_output(self, output):
        """Function to finish the forward pass"""
        output = self.relu(output)
        dense1 = self.fc(output)
        drop = self.dropout(dense1)
        output = self.output(drop)
        return output


    def forward(self, x, lengths, lengths_cpu):
        """ x is batch of padded sequences
        Lengths is a tensor of the lengths of each sequence in the batch
        Lengths_cpu is the input lengts on the cpu to increase speed"""
        x = self.embedding(x)

        """Pack padded sequence"""
        packed_embedded = pack_padded_sequence(x, lengths_cpu, batch_first=True,
                                               enforce_sorted=False)
        self.lstm.flatten_parameters()

        """Forward pass through LSTM"""
        output, (h,c) = self.lstm(packed_embedded)

        """ Unpack padding"""
        output, _ = pad_packed_sequence(output, batch_first=True)

        """pad_packed_sequence returns sequences with padding.
        Get indecies of the last value of the unpadded sequences with same
        dimension of output. In principle doing the same as output[:,-1,:] for
        batches with sequences of equal lengths.
        We just want the hidden states of the last output"""
        index = torch.sub(lengths,1)
        index = index.unsqueeze(-1)
        index = index.repeat(1,self.hidden_size)
        index = index.unsqueeze(1)


        """ Use the computed indecies to get last output per sample from the output """
        output = output.gather(dim=1, index=index).squeeze()
        output = self.process_output(output)
        """ Sigmoid layer is omitted as it is computed by the nn.BCEWithLogitsLoss() """
        return output


    def get_output(self, x, h=None, c=None):
        """
        Function doing the same as a forward pass without padded and packed sequences
        Returns hidden and cell state and adds a sigmoid layer.
        Used by L* extraction methods """

        out = self.embedding(x)
        self.lstm.flatten_parameters()
        if h == None and c==None:
            out, (h,c) = self.lstm(out)
        else:
            out, (h,c) = self.lstm(out, (h, c))
        out = out[-1,:]
        out = self.process_output(out)
        out = torch.sigmoid((out))
        return out, (h,c)


#=====================================================================================
#------------------------------ L* extraction methods -------------------------------

    def get_first_RState(self):
        """ Initiate an empty state """
        h = torch.zeros(self.n_layers, 1, self.hidden_size).to(self.device)
        c = torch.zeros(self.n_layers, 1, self.hidden_size).to(self.device)
        h = h.flatten()
        c = c.flatten()
        first_state = torch.cat((c, h), 0).tolist()

        # Initial class not important in our case as we do not use empy words
        classification = True
```

```python
        return first_state, classification


    def classify_word(self,word):
        """ Classify an import word """

        if word == "": # Handle empty word
            return True

        processed_word = torch.tensor([self.symbol_dict[w] for w in word],
                                      device=self.device)
        output, (h,c) = self.get_output(processed_word)
        val = output[0].item()
        classification = val > 0.5
        return classification


    def get_next_RState(self, state, char):
        """  Get next state and classification from a previous hidden and cell state.
        """

        """ Split state into hidden and cell states """
        h = torch.tensor(state[self.hidden_size*self.n_layers:],
                         device=self.device).unsqueeze(0)
        c = torch.tensor(state[:self.hidden_size*self.n_layers],
                         device=self.device).unsqueeze(0)

        """ Seperate h and c into layers """
        h = h.chunk(self.n_layers, dim=1)
        h = torch.cat(h, dim=0)
        c = c.chunk(self.n_layers, dim=1)
        c = torch.cat(c, dim=0)

        char = torch.tensor([self.symbol_dict[char]], device=self.device)

        """ Make prediction """
        out, (h,c) = self.get_output(char, h, c)
        h = h.flatten()
        c = c.flatten()
        next_state = torch.cat((c, h), 0)
        pred = out > 0.5
        classification = pred[0].item()

        next_state = next_state.tolist()
        return next_state, classification
```

Listing A.1: LSTM model code