Simplifying state machine models with sequence alignment for anomaly detection

Suo Xian Zhang

Technische Universiteit Delft

TU Delft

Simplifying state machine models with sequence alignment for anomaly detection

by

Suo Xian Zhang

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 30, 2024 at 12:30 PM.

Student number:	4660129	
Project duration:	September 1, 2023 – August 30, 2024	
Thesis committee:	Prof. dr. ir. S. Verwer,	TU Delft, supervisor
	Prof. dr. ir. A. Panichella,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

This thesis is the result of my academic journey, building upon the knowledge and experiences I have gained during my Master's in Computer Science. It reflects many months of researching, exploring relating topics, and carefully determining the direction I wanted to pursue for this research. Much time was devoted to studying and understanding state machine learning, Flexfringe, the characteristics of software logs and sequence alignment, along with extensive trial and error in during the implementation of the Needleman-Wunsch algorithm.

I would like to express my sincere gratitude to my supervisor, Sicco Verwer, for his fast reply when I expressed my interest on this topic on Project Forum and for introducing the idea of this thesis to me. His enthusiasm and insights sparked my interest in a topic that I was initially unfamiliar with. Writing this thesis was not an easy task for me, and I am thankful for his encouragement and patience throughout the entire process, especially for his willingness to explain concepts multiple times when I struggled to understand them. I also wish to thank my fellow master's students who participated in the weekly meetings, for sharing laughs, engaging in interesting discussions and providing mutual support.

Lastly, I would like to thank my parents for their trust and patience, and my friends for encouraging me and believing in me.

Throughout this research, I have gained not only a deeper understanding of state machine learning and its incredible possibilities, but also valuable insights into my own capabilities and work ethic.

I hope this thesis sparks the same interest in you that it did in me and provides you with valuable insights into the world of state machine learning.

Suo Xian Zhang
Delft, August 2024

Contents

1 Introduction **1**

1.1 Research Questions . . . 2

1.2 Contributions . . . 2

1.3 Outline . . . 3

2 Background **5**

2.1 Sequential data . . . 5

2.1.1 Parallelism . . . 6

2.2 Transition systems . . . 6

2.3 Petri-nets . . . 7

2.4 State Machine Models . . . 8

2.5 State Machine Learning . . . 10

2.5.1 Identifying Regular Languages . . . 11

2.6 State merging algorithms. . . . 11

2.6.1 Evidence driven-state merging (EDSM) . . . 12

2.6.2 ALERGIA . . . 12

2.7 Flexfringe . . . 13

2.7.1 Red-blue framework . . . 13

2.7.2 Evaluation functions . . . 14

2.7.3 Parameters . . . 15

2.7.4 Input Format . . . 15

2.7.5 Output format . . . 16

2.7.6 Prediction . . . 17

2.8 Comparing state machine models . . . 17

2.8.1 Labelled Transition System . . . 18

2.8.2 Structure comparison of LTS . . . 18

2.8.3 Language comparison of LTS . . . 19

2.9 Sequence alignment . . . 20

2.9.1 Distance measure . . . 21

2.9.2 Needleman-Wunsch algorithm . . . 22

2.9.3 Smith-Waterman algorithm . . . 24

2.9.4 Tree-Sequence Alignment . . . 24

2.10 Evaluation metrics . . . 25

3 Related Work **29**

3.1 Anomaly detection in logs . . . 29

3.2 Log-based behavioural differencing . . . 30

4 Methodology **33**
4.1 Data Exploration . . . 33
4.1.1 Data preprocessing . . . 33
4.1.2 Hadoop Distributed File System . . . 33
4.2 Modelling data with FSA . . . 35
4.2.1 Heuristic selection . . . 35
4.2.2 Sink count . . . 36
4.3 Sequence alignment . . . 36
4.3.1 Evaluation . . . 38
4.3.2 Classification . . . 38

5 Experiments **39**
5.1 DFA model . . . 39
5.2 Experiment 1: Scoring . . . 39
5.2.1 Static scoring . . . 40
5.2.2 Linear scoring . . . 40
5.2.3 Dynamic scoring . . . 40
5.2.4 Results . . . 40
5.3 Experiment 2: Parallelism . . . 44
5.3.1 Ignore-skip rule . . . 45
5.3.2 Results . . . 45
5.4 Experiment 3: Modifying the original model . . . 46
5.4.1 Results . . . 48

6 Discussion **51**

7 Conclusion **53**

A Appendix A **55**
A.1 Learned FSA . . . 56
A.1.1 HDFS with AIC and sink count=5 . . . 56
A.1.2 HDFS with AIC and sink count=10 . . . 57
A.1.3 HDFS with AIC and sink count=50 . . . 58
A.1.4 HDFS with AIC and sink count=100 . . . 59
A.1.5 HDFS with AIC and sink count=300 . . . 60
A.1.6 HDFS with AIC.ini . . . 61
A.1.7 HDFS with edsm.ini . . . 62

B Appendix B **63**
B.1 Confusion Matrices for predict . . . 63
B.2 Confusion Matrices for predictalign . . . 66
B.3 Confusion matrices for Needleman-Wunsch alignment . . . 68
B.3.1 AIC model, sinks=50 . . . 68
B.3.2 AIC model, sinks=100 . . . 70
B.3.3 AIC model, sinks=300 . . . 71
B.4 ROC curves . . . 71

1 Introduction

Current software systems have become increasingly complex. Software logs are generated and stored by computer systems, operating systems, applications, and other software components to track chronological records of activities, events, and behavior. Logs are crucial for system debugging, error diagnosis, system health monitoring, performance monitoring, problem troubleshooting, and security maintenance, often performed by experts in security operation centers. Logs can also detect potential abnormal behavior and help identify the timing and cause of such behavior by providing insights into when the issue first occurred, what caused it, and the differences between normal and abnormal behavior.

While logs are very useful, they come with certain drawbacks. First, logs can be very large, and they can be produced in large volumes. Second, logs often vary in format, increasing their complexity. As systems scale, the amount of information increases, making it more difficult for human experts to manually understand, analyze, and debug these systems and their logs. This increases time consumption and leads to a condition known as *alert fatigue*, where experts fail to respond to alerts due to the overwhelming volume they receive [1].

Several solutions have been proposed to combat this issue: (1) experts create *ad hoc scripts* to search for keywords such as ”error” or ”critical” to reduce workload, (2) use a method called *alert triaging* to prioritize alerts, or (3) enhance individual detectors to reduce generated security events [2]. However, these methods have proven insufficient for anomaly detection [3]. They are mainly effective in detecting *point anomalies*, where individual instances are considered anomalous compared to the rest of the data. Complex attacks involving combinations of events often go unnoticed while singular suspicious events are emphasized. *Contextual anomalies*, which occur when an instance is not anomalous on its own but is in a specific context, fail to be recognized. Another type of anomaly, more complex to find, is *collective anomalies* which are collections of related data that are anomalous concerning the rest of the data. The individual instances may not be anomalies, but their occurrence together makes them anomalous [4]. Therefore, a more effective approach for identifying an attack would involve analyzing a ”sequence” of events.

In many cases, one is also interested in multiple logs rather than a single log. The combination of logs originating from a different set of runs of the system at hand give insight into the context of evolution, malware analysis and testing and deployment, as these cases require comparative analysis rather than individual log analysis [5].

Research has been conducted on extracting various models from software logs to capture con-

textual and collective anomalies, with most using deep learning models [2][3][6][7]. However, these models are black boxes that provide no insight into the execution paths or decision-making processes. There lack in techniques that can extract underlying structures and significant patterns from log data for use in security management [8]. It is important that these techniques and models are unsupervised, as anomaly labels are often unavailable in practice. One approach that can effectively model behavior is the use of state machine models or Finite State Automata (FSA). These models use unlabeled data to represent system behavior and can visualize execution paths and context from large amounts of data, giving experts greater insight into understanding and identifying anomalous events.

There are two types of state machine models: *Deterministic Finite Automaton (DFA)* and *Non-deterministic Finite Automaton (NFA)*. DFAs need a source state and input symbol to transition into one particular next state and allow only one state transition per event. NFAs on the other hand, allow transitions into multiple next states, hence the name 'non-deterministic'. NFAs are more compact, but introduce an overhead associated with maintaining several states for one execution. DFAs allow faster executions due to their requirement of one state per traversal, making them better suited for anomaly detection.

However, modeling an automaton that captures all execution paths from large datasets often results in a model that is difficult to interpret. When attempting to model all available data, FSAs commonly suffer from a phenomenon known as *state explosion*, where the automaton becomes very large. Typically, a large automaton features (1) a few initial states that are frequently encountered and match many logs, with tails that are rarely visited, and (2) many parallel sequences.

To address the aforementioned issues and mitigate the risk of state explosion, this thesis will explore behavioral differencing of software logs involving sequence alignment on a compact representation of the system while identifying its the dominant and persistent behaviors.

This brings us to the research questions:

1.1. Research Questions

This brings us to the following research questions:

1. How effective is sequence alignment on a compact model of behavior in detecting flows that deviate from expected behavior?
2. What are the characteristics of parallel processes in Finite State Automata (FSA) and what rules can be added to the sequence alignment to effectively recognize these?
3. How effective are these rules in combination with sequence alignment on recognizing parallel processes in a compact model of behavior?
4. Can sequence alignment improve the learning process on a compact model?

1.2. Contributions

The main contributions of this thesis are:

1. **Describing sequential data** with an introduction of several models that capture sequential data.
2. **Explanation on state machine learning**, specifically, on state machine learning, merging algorithms for learning state machines, its use cases, the Flexfringe tool for automaton learning and comparing state machines.
3. **Introduction on sequence alignment** with background information and topic explanation of sequence alignment.

4. **The implementation of the Needleman-Wunsch algorithm**, modified for sequence to state machine model alignment. The algorithm is integrated into the Flexfringe codebase, along with documentation.
5. **Evaluation on a large dataset** demonstrating the capability to perform behavioral differencing effectively.

1.3. Outline

This thesis is outlined as followed:

- *Chapter 2* provides an overview of the background topics necessary for understanding the problem statement and the approach taken in this thesis. It defines sequential data and parallelism, and introduces the the models used to describe behavior. Additionally, this chapter explores state machine learning and offers a detailed explanation of sequence alignment.
- *Chapter 3* reviews related work on anomaly detection on log data and the use of Finite State Automata for anomaly detection.
- *Chapter 4* explores the dataset used in this thesis and explains the Needleman-Wunsch sequence alignment algorithm and the evaluation methods employed.
- *Chapter 5* presents three experiments designed to answer the research questions.
- *Chapter 6* presents a discussion on the findings, their implications and limitations.
- *Chapter 7* presents the final concluding remarks.

2
Background

This section begins with a detailed explanation of sequential data. It then introduces three models commonly used to describe behavior: Transition Systems, Petri Nets, and State Machine Models, with a specific focus on Finite State Automata (FSA). Since this thesis employs FSA, the concept of State Machine Learning will be explained, followed by an introduction to FlexFringe, a tool for automaton learning. Subsequently, the concept of sequence alignment will be introduced and two alignment algorithms are defined. Finally, relevant evaluation metrics will be discussed.

2.1. Sequential data

Software logs typically contain data with entries containing events, actions or behavior from a system in a chronological order. Each log entry includes details such as timestamps, event types, user actions, system states, error messages, and other relevant information. This sequential data is vital for monitoring, debugging, and analyzing system behavior. Every sequence of events in a log is also known as a *trace*.

Looking at individual events allows the detection of *point anomalies*, which are isolated anomalous events. However, *contextual*, events that are anomalous given their context, and *collective anomalies*, groups of events that are anomalous in relation to the rest of the data, can only be detected when the sequence of events is investigated. Therefore, a sequence of messages often provides a better indication of anomalies than individual messages [3]. Investigating sequences allow for:

- *Contextual Information*: Sequences provide context that individual messages lack. For instance, the absence of a commit message in the sequence indicates an issue that individual messages do not reveal.
- *Anomaly Detection*: Some anomalies, such as silent failures or incomplete operations, are only detectable when considering the entire sequence of events.
- *Causal Relationships*: Understanding the cause and effect between different events is crucial for diagnosing problems, which is only possible by analyzing sequences.
- *Pattern Recognition*: Certain issues manifest as specific patterns of events, which can only be identified by looking at the sequence as a whole.

2.1.1. Parallelism

In complex systems, many components and services operate concurrently, interacting independently while performing different tasks, this is known as parallelism. Parallelism is a common occurrence in modern computing systems, arising from multi-threading or simultaneous execution of multiple processes. Some examples of parallelism are:

1. *Concurrent Processes:* Multiple processes executing at the same time, often on separate processors.
2. *Multithreading:* Multiple threads within a single process running in parallel, sharing the same resources but executing independently.

These complex systems record logs, which contain the events that occurred concurrently. While logs are recorded sequentially, parallelism can cause events to appear out of order. Parallelism in software logs can look like:

1. *Interleaved events:* Log entries appear interleaved or interwoven, based on when the events are recorded while processes or threads execute concurrently.
2. *Simultaneous operations:* Log entries with the same timestamp, indicating parallelism, as the events are occurring concurrently.
3. *Fork and Join patterns:* Log entries of a single process may be interrupted by the log entries from another process and continue afterwards.
4. *Repeated or Overlapping entries:* Several instances of the same log entries appear, but are executed by different processes.
5. *Out of order logs:* Operations from parallel processes might appear out of order without clear sequence.

When analyzing software logs for anomaly detection or behavioral differencing, parallelism can introduce complexity. Learning behavior that involves parallelism with state machine models can be challenging. New software logs may contain log entries in varying order due to parallelism, while essentially representing behavior that the model actually already knows but cannot recognize. Parallelism can also cause state machines to become overly complex, or ”blow up”, which makes analysis difficult. Effectively modeling and detecting parallelism can be of great use in behavioral differencing and identifying anomalies.

Next, three systems designed to translate sequential data into models that model their behavior are introduced and defined.

2.2. Transition systems

A transition system is the most basic mathematical model used to describe the behavior of systems. The formal definition of a transition system is as follows:

Definition 1. *Definition of a Transition system, as defined in [9]*

A transition system is a quadruple TS = (S, →, s_0, F), where:

1. S is a set of *states*.
2. $\rightarrow \subseteq S \times A \times S$ is the *transition relation*, written $s \xrightarrow{a} s'$ for $(s, s') \in \rightarrow$.

3. $s_0 \subseteq S$ is the *initial state*.

4. F is the set of *final states*, where $F \subseteq S$.

It is particularly useful for modeling concurrent and distributed systems. The key components of a transition system are:

- *States*: These represent the different configurations or conditions of the system.
- *Transitions*: These are the changes from one state to another, often triggered by events or actions.
- *Labels*: Transitions can be labeled with actions or events that cause the transition. In this case the model is a *Labelled transition system*.

Transitions starts in the initial state s_0. An *execution sequence* is the path taken in the transition system, from the initial state to any other state. A path only terminates successfully if it ends in $s' \in F$. Otherwise, if the path ends in $s \notin F$ with no outgoing transitions, the system has reached a *deadlock*. *Livelock* happens when some transitions are still enabled but it is impossible to reach a $s \in F$. Higher-level models can often be transformed to transition systems.

While transition systems are simple and effective for modeling processes, they have limitations in expressing concurrency as transition systems typically represent processes as sequences of discrete states and transitions between them, which may not capture the simultaneous occurrence of similar actions or events. For instance, in system logs, actions like file writes or database accesses by different processes may appear very similar and be seen as occurring in parallel. However, transition systems may struggle to differentiate and model these parallel actions distinctly, potentially oversimplifying the behavior of the system. This limitation highlights the need for more expressive models, such as Petri nets or state machine models, which can better handle and represent parallelism in system behavior. This brings us to the next model: Petri nets.

2.3. Petri-nets

A Petri-net, or place/transition net (PT) net, is a mathematical model for languages, capable of modelling concurrent systems. Petri-nets can be treated as automata when considered as formal automata or as generators of formal languages. The formal definition of a Petri-net is as follows:

Definition 2. *Definition of a Petri-net*

A Petri-net is a tree-tuple N = (P,T,F), where:

1. *P* is a finite set of *places*.
2. *T* is a finite set of *transitions*, such that $P \cap T = \emptyset$.
3. *F* ⊆ *(P* × *T)* ∪ *(T* × *P)* is a set of directed arcs, also known as *flow relation*.

The network structure remains static, while *tokens* have the capability to traverse through the network. These *tokens* serve as markers for movement and can reside in *places*, often depicted by black circles within the place. For instance, observe the token residing in the *start* place depicted in Figure 2.1. The tokens can be moved by *firing* a transition within the net. The following rules exist for firing a transition [10]:

1. Transitions can be fired only if they are *enabled*, which is the case when all of its input places have a token in it. In Figure 2.1, the *'decide'* transition can only be fired if both input places *c3* and *c4* are marked.

2. An enabled transition may also not fire depending whether or not the event takes place.

3. Upon activation, the transition consumes the tokens from the input places and produces new tokens in the output places associated of the transition, see Figure 2.2. When firing two transitions disables the other, they are said to be *in conflict*.

A transition lacking any input place is referred to as a *source transition*, an example is the *start* place in Figure 2.1. This type of transition remains enabled under all conditions. Conversely, a transition without an output place is called a *sink transition*, exemplified by the *end* place in Figure 2.1. Despite its ability to consume tokens, a *sink transition* does not produce new tokens. Additionally, when a place serves as both the input and output of a transition, it is referred to as a *self-loop*. Notably, a *pure* Petri-net is devoid of self-loops.

Transitions model *concurrency* with multiple outgoing arcs. An example of this is the *'register request'* transition in Figure 2.1; after firing the *'register request'* transition, tokens are produced in *c1* and *c2* which allows the execution of the sequences *[start, register request, examine thoroughly]*, *[start, register request, examine casually]* and *[start, register request, check ticket]*.

Choice is modelled with a place with multiple outgoing arcs, see *c5* in Figure 2.1, it means either transitions *'pay compensation'* and *'reject request'* can be fired.

The state of a marked Petri-net is determined by the distribution of tokens within it, which is referred to as its *marking*. A marked Petri-net is a pair *(N,M)*, where $N = (P, T, F)$ is a Petri-net and where $M \in \mathbb{B}(P)$ is a *multi-set* over *P* denoting the *marking* of the net. The set of all marked Petri-nets is denoted $\mathcal{N}$.

While Petri nets provide a structured approach for modeling concurrent systems, state machine models focus on discrete, sequential behavior and transitions and will be further explained in the next section.

2.4. State Machine Models

State machine models are abstract computational models used to describe systems that transition between a finite number of states based on inputs or events. These models consist of states, transitions between states, an initial state, a set of final or accepting states. State machine models are employed to represent and analyze the behavior of both software and hardware systems, making them fundamental in fields like computer science, engineering, and control systems. As mentioned in the introduction, they can be *deterministic*, where each state has a unique transition for a given input, or

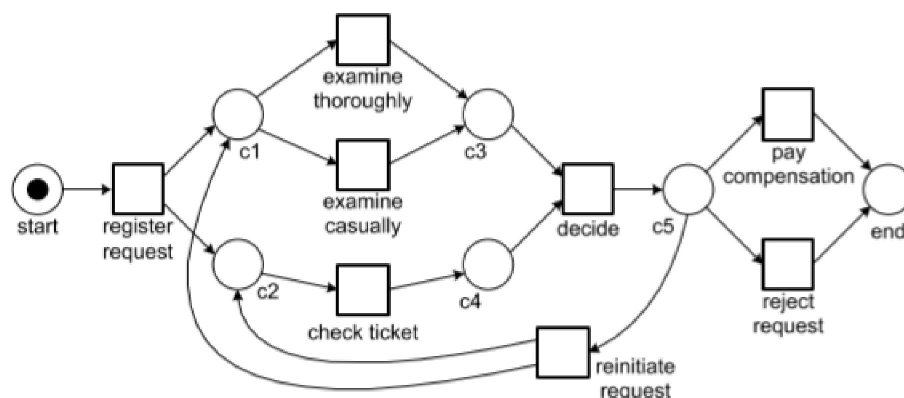
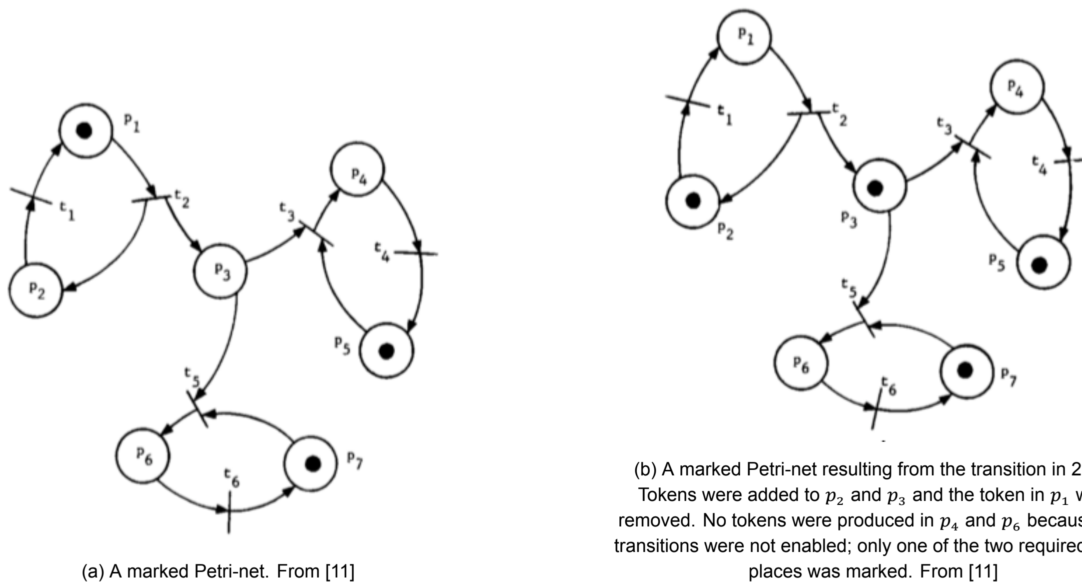


Figure 2.1: A Petri-net for compensation requests. From [9].



(a) A marked Petri-net. From [11]

(b) A marked Petri-net resulting from the transition in 2.2a. Tokens were added to p_2 and p_3 and the token in p_1 was removed. No tokens were produced in p_4 and p_6 because the transitions were not enabled; only one of the two required input places was marked. From [11]

Figure 2.2: Two markings presenting two different states in a Petri-net

nondeterministic, where multiple transitions for a given input are possible. Overall, state machine models provide a structured way to model dynamic system behavior, assisting in the design, verification, and understanding of complex systems.

Variants of state machine models include *Mealy and Moore machines*, which generate outputs based on state transitions, and *finite state automata (FSA)*, which are used for pattern recognition and language processing. FSA are computational models used to design and describe the behavior of systems that can be in different states and undergo state transitions, see Definition 3 [12]. An FSA can be represented by a *state transition diagram*, a directed graph whose vertices represent the states Q and whose edges represent the state transitions δ. Each edge is labelled with the input and output related to the transition, see Figure 2.3.

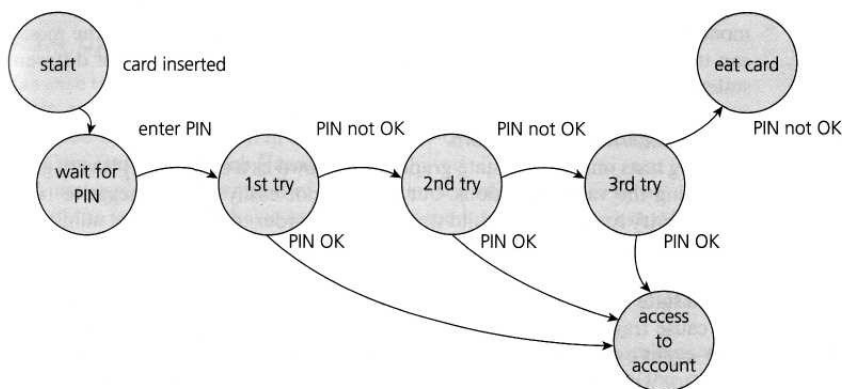


Figure 2.3: State transition diagram example for an ATM, taken from [13].

An FSA processes an input sequence symbol by symbol, transitioning to the next state based on the defined transition function δ. Upon completing the parsing of the entire input sequence, the FSA accepts the sequence if it is in an accepting state; otherwise, it rejects it.

Definition 3. *Defintion of a FSA*

A FSA is a quintuple M = (Q, q_0, Σ, F, δ), where:

1. Q is a finite set of *states*.
2. Σ is a finite set of input symbols, known as the *alphabet*.
3. q_0 is the *starting state*, where $q_0 \in Q$.
4. F is the *set of accepting or final states*, where $F \subseteq Q$.
5. δ is the *state transition function*: $Q \times \Sigma \to Q$. $\delta(q, a)$ gives the next state when the automaton is in state q and receives input symbol a.

FSAs can be divided into *Deterministic Finite Automaton (DFA)* and *Non-deterministic Finite Automaton (NFA)*.

Deterministic Finite Automaton (DFA) In a DFA, at any given time, there is only one possible transition from a state for each possible input. The transition is determined uniquely by the current state and the input. The definition of DFAs is similar to FSAs, but the crucial distinction here is that for any state *q* and input symbol *a*, there is at most one possible next state δ(q,a). In other words, it is deterministic.

Non-deterministic Finite Automata (NFA) In an NFA, there can be multiple possible transitions from a state for a given input. The machine may be in multiple states simultaneously, and the transition is not uniquely determined by the current state and input. Due to the overhead involved in maintaining multiple states for a single execution in NFAs, and considering that DFAs demonstrate faster execution rates by requiring only one state traversal per character, this thesis will utilize DFAs.

2.5. State Machine Learning

State machine learning is a machine learning method that abstracts models in the form of state machine models from data. These state machine models are generally in the form of DFA. This problem is also known as the problem of identifying or learning a DFA.

The goal of state machine learning is to identify the most compact DFA that aligns with a provided set of labelled examples [14]. The size of the DFA is determined by its number of states. This pursuit of minimalism is driven by *Occam's razor principle*, which states that among competing explanations, the simplest one is typically the most preferable. Thus, a smaller DFA is favoured as it offers a more straightforward and concise explanation.

State machine learning generally follows three steps as illustrated in 2.4. In the first step, data is collected. In the second step, the *Prefix Tree Acceptor (PTA)*, which is the initial DFA is constructed from the input samples S collected in the first step. Formally, the constructed PTA is consistent with the input sample S, meaning $S_+ \subseteq L(A)$ and $S_- \cap L(A) = \emptyset$ In the final step, the underlying automata are learned by merging compatible pairs of PTA states [15]. Two states q and q' can be merged into a new state q'' with the incoming and outgoing transitions from q and q' if both states are *consistent*, meaning they are both accepting and rejecting the same transitions [16]. Furthermore, the *determination process* is applied during every merge. During this process, when q'' is the source of two transitions with the same symbol, the states are known to be *non-deterministic*, in which case, the target states of these transitions are also merged. This process continues until there are no more non-deterministic choices left and no more consistent merges. This reduces the size of the automaton. For this step, different algorithms can be used.

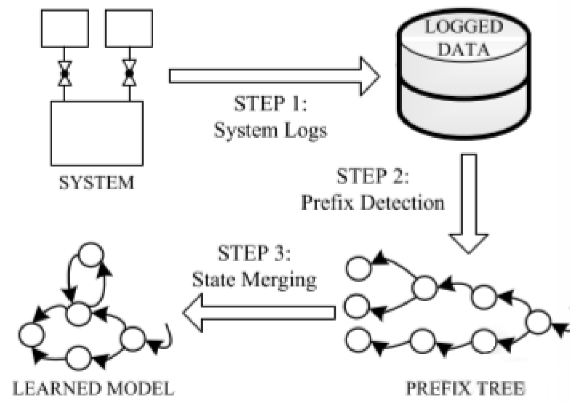


Figure 2.4: State merging approach for learning automata [15]

2.5.1. Identifying Regular Languages

Regular languages are a class of formal language defined by a set of rules known as regular expressions. These languages are used to describe patterns or sequences of characters within a larger text or input stream. Languages can be represented by FSA.

A FSA M can describe a language as follows: let ω be a string, M is said to accept string ω if M starts from the starting state, follows transitions corresponding to the symbols in ω, and ends up in an accepting or final state. In other words, M *recognizes* language L if M accepts all strings ω that are in L. A language is regular if there is an FSA that recognizes it. The language of an automaton A is represented as $L(A)$.

Let Σ^* be the set of all finite strings over an alphabet $L \subseteq \Sigma$. Given a regular language $L \subseteq \Sigma^*$, $S^+ \subseteq L$ are strings in L (positive examples) and $S^- \subseteq \overline{L}$ are strings not in L (negative examples). The combination of S^+ and S^- form a complete set S. As elucidated in [17], to identify the regular language, the limit L must be found, where new examples to S only result in a finite number of changes. Examples from S^- are significant for rejecting a language L' that differs from L only in the strings it in includes $L' - L$. There may exist languages where the only distinction between them lies in the strings they contain. Nonetheless, it is common for negative examples to be lacking. A set S containing only S^+ is known as a *positive sample* or *text*.

Given a language L, the minimum DFA generating L is called the *canonical acceptor M(L)*. If the PTA T is defined for a finite complete set S of L, the problem of identifying L is then simplified to finding a partition of the nodes in T. The language L_0 accepted by T is the positive part of S, so $L_(0) = S^+$. The goal is to find a partition that does not include negative examples from S^-, ensuring that the language of the partition remains unchanged even with the addition of new examples to S. To find the partition, a merging process can be deployed. When S^- is large enough to reject any incorrect merges, $M(L)$ will be the final output. In other words, once S^+ and S^- reach a certain size, the hypothesis automaton remains unchanged, giving a successful identification of L.

2.6. State merging algorithms

The issue of finding the smallest consistent DFA can be very complicated. The optimization variant of the problem of finding a consistent DFA of fixed size has been shown to be NP-complete [14].

Nevertheless, there are multiple state machine learning algorithms available for obtaining these DFA models. These algorithms can operate either *online*, where additional examples can be requested during the learning process, or *offline*, where only a given dataset is utilized. The algorithm leverages both *positive* and *negative* examples.

2.6.1. Evidence driven-state merging (EDSM)

EDSM [18] starts with a PTA for the training data and merges compatible states. States are defined as compatible when no suffix leads from them to different labels. It is critical for the algorithm's early merge decisions to be correct, as a merge introduces new constraints on future merges. Incorrect decisions at the beginning will create wrong constraints leading to incorrect decisions later.

When a state merging algorithm is applied to sparse training data, each merge is often based on an optimistic guess. Therefore, a good strategy is to first perform the merges that are supported by the most evidence, which is why the term 'evidence' is included in the algorithm's name. This will minimize the snowballing effect.

The initial hypothesis is the PTA that represents the training set. The algorithm computes the score for every pair of nodes in the hypothesis. While the merges are valid, it performs the highest scoring one, otherwise, it stops. It proceeds this process until there are no valid merges left.

To merge nodes, the partition of hypothesis nodes into *equivalence classes* must be determined. Two candidate nodes are of equivalent class if they are: 1) equivalent, with regards to their transition function, and 2) conform to the *determinization rule*, which states that children of equivalent nodes must also be equivalent. A find/union data structure keeps track of sets that are known to be equivalent. During every equivalence check, the structure of the two nodes is used to determine whether they can be merged. The two nodes also have to be consistent, which can be determined with a statistical test or distance computation based on their future sequences. If two nodes can be merged, it merges the equivalence classes linked to every member of the input set, then recursively invokes itself on each of the sets of children of the newly unified equivalence class. Otherwise, the procedure returns.

With the equivalence classes and the labels, a merge score can be computed. The *merge score* is $-\infty$ if there are conflicting labels in the class, 0 if there are no labels in the class, otherwise, it is the number of labels minus one.

If the merge is also valid by a non-negative score, a new hypothesis is constructed reflecting the merge. The new hypothesis will have one state per equivalence class. The merging process stops when there are no valid merges left.

2.6.2. ALERGIA

Alergia [17] merges states based on the probability of symbols. It takes advantage of the fact that the probability of appearance of every string follows a well-defined distribution and learns a PTA from a set of positive data and compensates for the absence of negative data. The algorithm merges states if the resulting automaton is in line with the observed frequencies of the data.

The algorithm creates a PTA from the input data and evaluates at every state the relative frequencies of the outgoing edges. The probabilities for ending in every state or transitioning to another state are computed based on 1) the number of strings that arrive at that state, whether it is termination or passing through, 2) the number of strings that end in that state, and 3) for each symbol the number of strings that pass through and then transition to another state using that particular symbol.

Two states are said to be equivalent if their outgoing transition probabilities for every symbol are the same as well as their final states. Due to statistical fluctuations in data, the equivalence between states is accepted within a confidence range α. Compatibility between two states are evaluated with the Hoeffding bound [19], see 2.1, which gives a confidence range for a Bernoulli variable where p is the probability, f is the observed frequency out of n tries:

$$\left|p - \frac{f}{n}\right| < \sqrt{\frac{1}{2n} log \frac{2}{\alpha}} \text{with probability larger than} (1 - \alpha) \tag{2.1}$$

Two states will be different if the two estimated probabilities are more than the sum of confidence ranges, see 2.2, where n and n' are the number of strings arriving at each state and f and f' are the number of strings ending in each state. Compatible and equivalent states are checked recursively.

$$\left|\frac{f}{n} - \frac{f'}{n'}\right| < \sqrt{\frac{1}{2n} log \frac{2}{\alpha}} (\frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n'}}) \quad (2.2)$$

Alergia ensures that the order of states is preserved during the merging process as well as the deterministic nature of the automaton. Furthermore, the frequencies and numbers are recalculated at every merge.

2.7. Flexfringe

Flexfringe is an automaton learning tool capable of learning non-probabilistic DFA and probabilistic DFA (PDFA). It is originally known as the DFASAT algorithm [14]. The algorithm is based on the red-blue framework [18].

The goal is to find the smallest DFA A that is consistent with $S = S_+, S_-$ such that $S_+ =\subseteq L(A)$ and $S_- \subseteq \Sigma * \backslash L(A)$, where $\Sigma*$ is the set of all strings. It starts with a PTA and iteratively combines similar states until no similar states can be found. The similarity of their future behavior is determined using a Markov property or a Myhill-Nerode congruence. To find the smallest consistent (P)DFA, Flexfringe uses the Red-Blue framework (EDSM).

2.7.1. Red-blue framework

The algorithm [18], like most, starts with a PTA. The root is coloured red, its children blue and all other nodes white. The following cases apply:

1. An arbitrary connected graph of red nodes that cannot be merged.
2. All non-red children of red nodes are blue.
3. Blue nodes are the roots of isolated trees.

The algorithm assumes the red core is correctly identified and the blue states are merge candidates. Three actions can be performed:

1. Compute the score for merging a blue/red pair.
2. Color a blue node to red if it cannot be merged with any red node.
3. Merge a red node with a blue node.

These three cases and three actions can be used in different algorithms. One of those follows the following steps:

1. Evaluate all red/blue merges.
2. If there exists a blue node that cannot be merged with any red node, the lowest blue node will be colored red and continue from step 1.
3. Otherwise, the highest red/blue merge will be performed and then continue from step 1.

The algorithm is complete as it produces a DFA that is smaller than the starting PTA and it is consistent with the input data [20].

2.7.2. Evaluation functions

As mentioned before, a statistical test or distance computation is required during every merge to check whether it is consistent, this is known as a *consistency check*. Consistency checks are implemented in the *evaluation function*. An evaluation function also implements a *score*, which provides means to determine which merge from a set of potential merges should be performed first. Flexfringe has several evaluation functions already implemented:

1. **Alergia** [17] relies on the Hoeffding bound for the consistency checks. For each potential merge, it checks for significant differences in the outgoing symbol distributions, preferring merges that combine the most states during determinization. Greater differences indicate greater similarity in distributions.

2. **Likelihoodratio** [21] computes a single test for the entire merge procedure, including determinization. It aims to address a possible pitfall Alergia may have. In Alergia, when large numbers of states have to be merged in the determinization, the possibility of a small number tests failing is high since each pair of states is tested individually. This prevents merging and results in a larger PDFA. The likelihoods and number of transitions of the PDFA before and after the merge are computed and evaluated [20]. The ratio is used to determine if the merged model outweighs the decrease in likelihood. A higher score means the decrease in likelihood is less important, indicating that the model before and after merging are more similar with regards to their distributions.

3. **MDI** [22] computes the likelihood and number of transitions of the models before and after merging, similar to likelihoodratio, and calculates the Kullback-Leibler divergence from the difference to the distribution in the original data sample. A merge is considered inconsistent if the distance is too large. MDI aims to address another potential pitfall of Alergia by providing the ability to bound the distance of the learned PDFA from the data sample.

4. **Akaike's Information Criterion (AIC)** makes a trade-off between the number of transitions and likelihood in a model [23]. It aims to minimize the number of parameters minus the loglikelihood. All merges that decrease the AIC are considered consistent. Essentially, it measures whether the reduction in transitions after the merge is greater than the decrease in loglikelihood. It is commonly used for probabilistic models.

5. **EDSM** uses positive and negative data to determine equivalence classes and merges these based on a score: the sum of the number of non-conflicting labels of the equivalence classes of the states to be merged [18].

6. **Overlap driven** is based on EDSM with a a modified heuristic [16] and is the winner of the StaMInA competition [24]. It is derived from DFASAT with two key changes: 1) it considers that pairs of states with outgoing transitions labeled with identical symbols are more likely to be equivalent than in smaller alphabets, and 2) it includes stochastic elements in the selection process of states to be merged, rather than always following a fixed rule or pattern. The heuristic favours the merging of states with a high degree of overlap in the symbols of their outgoing transitions, which can be measured by counting the amount of merged between states that can be reached by positive examples. This evaluation method is useful for large alphabets and sparse data sets.

Several .ini files are provided with commonly used settings[1]: `aic.ini`, `alergia.ini`, `edsm.ini`, `likelihood.ini`, `markov_chain.ini`, `overlap.ini`, `rti.ini`, `spdfa.ini`.

[1] `https://github.com/tudelft-cda-lab/FlexFringe/tree/main/ini`

2.7.3. Parameters

Flexfringe requires the user to set the `heuristic_name` and `data_name`. Moreover, it has several parameters available allowing the user to optimize the tool according the requirements of the users. Some relevant parameters are defined in Table 2.1. These parameters are essential for fine-tuning

Parameter	Definition
`heuristic_name`	The name of the merge heuristic to use. Several heuristics are implemented in Flexfringe and can be used; default: count_driven.
`data_name`	The name of the merge data class to use; default count_data.
`state_count`	The minimum number of positive occurrences of a state for it to be included in overlap/statistical checks. Used to ignore parts of the tree for which very few arrivals are available; default=25.
`symbol_count`	The minimum number of traces required to trigger a transition in a state for it to be considered for state merging; default=10;
`sinkson`	Determines whether sinks are allowed in the model or not. Sinks are used to represent the model in a more compact manner by grouping states with a low count, count can be determined by the parameter `sinkcount`, into a single sink node; default=0;
`sinkcount`	The minimum count a states should have to before being merged into a sink state. If the number of arrivals to a state is less than `sinkcount`, it is merged into a sink; otherwise, it remains a normal state; default=10;
`largestblue`	Determines whether only the most frequent candidate (blue) states with any target (red) state, rather than all candidate states; default=0.
`mergelocal`	Determines whether only local merges up to APTA distance k are performed; default=0;
`lowerbound`	Determines whether a minimum value is used for the heuristic function. The minimum value is determined with `lowerboundval`. When turned on the merger can prefer coloring a state red rather than performing a bad merge; default=0.
`lowerboundval`	Determines the minimum value is used for the heuristic function; default=0.
`printwhite`	Prints the white states (states not considered for merging) in the .dot file; default=0;
`printblue`	Prints the blue states (candidate states) in the .dot file; default=1;

Table 2.1: Flexfringe parameters and their definition

the learner's outcome. The most frequently used parameters are `symbol_count` and `state_count` parameters. It's important to note that the choice of algorithm for learning depends on the available input data.

2.7.4. Input Format

Flexfringe takes a .txt file as input. The input is required to be formatted following the *Abbadingo formatting*[18]:

1. The first line contains the number of traces and the alphabet size.
2. The following lines are the traces.

3. All traces start with the type and trace length.

See Figure 2.5 for an example of an input file according the Abbadingo format.

```
1   4855 14
2   1 19 5 5 5 22 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
3   1 13 22 5 5 5 11 9 11 9 11 9 26 26 26
4   1 21 22 5 5 5 26 26 26 11 9 11 9 11 9 2 3 23 23 23 21 21 21
5   1 13 22 5 5 5 11 9 11 9 11 9 26 26 26
6   1 31 22 5 5 5 26 26 26 11 9 11 9 11 9 4 3 3 3 4 3 4 3 3 4 3 3 23 23 23 21 21 21
7   1 31 22 5 5 5 26 26 26 11 9 11 9 11 9 3 3 4 3 4 3 3 3 4 4 3 3 23 23 23 21 21 21
8   1 19 5 22 5 5 26 26 11 9 11 9 11 9 26 23 23 23 21 21 21
9   1 23 22 5 5 5 26 26 26 11 9 11 9 11 9 4 4 3 2 23 23 23 21 21 21
10  1 19 5 22 5 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
```

Figure 2.5: First few lines of the HDFS training data in Abbadingo format. The first line contains the number of entries and alphabet size, the lines beneath are all log entries start starting with the message type, in this case it is '1', followed by the sequence of events.

2.7.5. Output format

Flexfringe outputs the model as JSON file. The JSON file contains four properties: types, alphabet, nodes and edges. Nodes presents individual states in a graph or network. Each item in nodes contains various attributes that describe its properties, see Table 2.2.

Field	Value
id	0
source	-1
label	fin: 0:0 , path: 0:4855 ,
size	4855
level	0
style	""
isred	1
issink	0
isblue	0
trace	0 0
data.final_counts	{ "0":0 }
data.path_counts	{ "0":4855 }
data.symbol_counts	null
data.total_final	0
data.total_paths	4855
data.trans_counts	{ "11":"0", "18":"0", "2":"0", "21":"0", "22":"1258", "23":"0", "25":"0", "26":"2", "3":"0", "4":"0", "5":"3595", "9":"0" }

Table 2.2: Node Item

Some of the properties are explained in more depth:

1. *id*: unique identifier for every node.
2. *source*: source of the edge that node is the destination of.

3. *label*: the label presented in the node.

4. *size*: amount of edges that the node is the destination of.

5. *level*: level of the node.

6. *trace*: trace that leads to the node.

7. *data*:

   - *total final*: how many traces end in the node.
   - *total paths*: how many traces pass the node.
   - *trans counts*: counts for every symbol in the alphabet how often an edge leaves the node with the symbol.

Every item in edges represents connections between the nodes, see Table 2.3. The id of an edge is structured such that it is *source node_target node*. Name is the symbol that the edge takes.

Field	Value
`id`	`"0_1"`
`source`	`"0"`
`target`	`"1"`
`name`	`"5"`
`appearances`	`""`

Table 2.3: Edge Item

2.7.6. Prediction

Flexfringe includes two options for prediction: *predict* and *predict_align*. *Predict* checks whether the data conforms to the learned model, whereas *predict_align* allows tracing model alignment by jumping to any state in the model necessary when a mismatch is encountered.

To run *predict_align*, an APTA file with the learned automaton has to be provided. The function returns a '.result' file that contains all the traces with their aligned state sequence, score sequence, alignment, number of misaligned states, sum score, mean score and min score. State sequence is an array with at each index the unique ID of the associated state corresponding to the index in the trace. If alignment is not possible, the trace will have an empty aligned state sequence and a sum score of either '0' or '-inf'.

2.8. Comparing state machine models

Instead of stating whether one model is equivalent to another model, the ability to compare two models, i.e. quantify the difference and equivalence, is important for a range of scenarios. Two FSAs can be compared quantitatively with regard to their structure and their language.

- Structure - states and transitions that the FSA constitutes of

- Language - sequences of symbols that are accepted by the FSA i.e. what is accepted by their languages

2.8.1. Labelled Transition System

As introduced in Section 2.2, the simplest interpretation of a state machine is a transition system. A labeled transition system (LTS) extends the transition system by adding labels to the transitions. A LTS reduces an FSA to a set of states and transitions, where each transition is labelled by a symbol, see Definition 4. Using a LTS, an FSA can be compared with regards to their structure and language as mentioned previously.

Definition 4. *Definition of a LTS*

A labelled transition system is a quintuple tuple $(S, \Sigma, \rightarrow, s_0, F)$, where

1. S is a set of *states*.
2. Σ is a finite set of *labels* (actions or events).
3. $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation*, written $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$
4. $s_0 \in S$ is the *initial state*.
5. F is the set of *accepting* or *final states*, where $F \subseteq S$. For each $s \in F$ there is no $a \in \Sigma$ and $s' \in S$ such that $s \xrightarrow{a} s'$.

Similar to a FSA, a LTS can be deterministic and non-deterministic. However, the difference between the two lies in their formal definitions and applications. FSAs model machines while LTSs model behavior. Moreover, in LTSs the set of states Q does not need to be finite and final states must have no outgoing transitions [25]. The latter requirement indicates that a final state represents a successful termination of the process.

2.8.2. Structure comparison of LTS

To compare the structure of two models, one model will be compared to the reference model. Added and removed states have to be identified.

LTSDiff

LTSDiff is an algorithm proposed in [26] that finds the exact difference, i.e. states and transitions, between two LTSs using the computation scores that measure the similarity of individual pairs of states. The score is computed by matching up the surrounding states and transitions, which is a recursive process.

In the first step of the computation, the *local similarity*, i.e. overlap of immediate surrounding transitions, is calculated. The local similarity of two states is computed by dividing the number of *overlapping adjacent transitions* by the *total number of adjacent transitions*. Given an LTS A and B, for each label $\sigma \in (\Sigma_A \cup \Sigma_B)$ the number of matching transitions from states $a \in Q_A, b \in Q_B$ is counted in terms of the number of individual pairs of target states that can be reached by matching transitions, the similarity score can be calculated as follows:

$$Succ_{a,b} = \{(c, d, \sigma) \in Q_A \times Q_B \times (\Sigma_A \cup \Sigma_B) | a \xrightarrow{\sigma} c \wedge b \xleftarrow{\sigma} d\} \tag{2.3}$$

For outgoing transitions, see 2.4 , where L stands for local. This function calculates the set of all matching pairs of target states and transition labels in relation to outgoing transitions.

$$S^L_{Succ}(a, b) = \frac{|Succ_{a,b}|}{|\Sigma^{out}_A(a) - \Sigma^{out}_B(b)| + |\Sigma^{out}_A(a) - \Sigma^{out}_B(b)| + |Succ_{a,b}|} \tag{2.4}$$

The expression $|\Sigma_A^{out}(a) - \Sigma_B^{out}(b)| + |\Sigma_A^{out}(a) - \Sigma_B^{out}(b)|$ calculates the number of outgoing transitions from both states that are mismatched.

The set of matching incoming transitions is defined as follows:

$$Prev_{a,b} = \{(c,d,\sigma) \in Q_A \times Q_B \times (\Sigma_A \cup \Sigma_B) | c \xrightarrow{\sigma} a \wedge d \xrightarrow{\sigma} b\} \tag{2.5}$$

$$S_{Prev}^{L}(a,b) = \frac{|Prev_{a,b}|}{|\Sigma_A^{inc}(a) - \Sigma_B^{inc}(b)| + |\Sigma_A^{inc}(b) - \Sigma_B^{inc}(a)| + |Prev_{a,b}|} \tag{2.6}$$

The pairs of states with the highest scores are chosen to be *key pairs*, which serve as reference points to calculate the differences between the two models.

In the second step the *global similarity*, i.e. similarity of target and source state of these transitions, is calculated. For every pair of adjacent transitions that match, the similarity of the source and target states of these transitions has to be incorporated in the final similarity score; if they are similar, the score should be higher and vice versa. The following equation extends the local similarity scoring algorithm in 2.8 by aggregating the similarity score for every successive matched pair of transitions and takes into account the distance of source/target pairs:

$$S_{Succ}^{G1}(a,b) = \frac{1}{2} \frac{\sum_{(c,d,\sigma) \in Succ_{a,b}} (1 + kS_{Succ}^{G1}(c,d)}{|\Sigma_A(a) - \Sigma_B(b)| + |\Sigma_A(b) - \Sigma_B(a)| + |Succ_{a,b}|} \tag{2.7}$$

This formula is used to create a set of linear equations where each variable corresponds to a 2.7 for a specific pair of states $(a,b) \in Q_A \times Q_B$. This system can be solved for the forward and inverse direction, producing $S_{Succ}^{G}(a,b) + S_{Prev}^{G}(a,b)$ for every pair of states forming:

$$S(a,b) = \frac{S_{Succ}^{G}(a,b) + S_{Prev}^{G}(a,b)}{2} \tag{2.8}$$

This system of linear equations is used to select pairs that are most likely to be equivalent. Only the pairs above a threshold *t* and pairs where the best match is at least *r* time as good as any other match are added to the set of key pairs. Afterwards, it links each feature in one model to its counterpart in the other, beginning with landmarks and then examining the surrounding vicinity. The algorithm continues this process until no more pairs of states or transitions can be found, leaving behind only the remaining differences. See **??** for the formal algorithm.

2.8.3. Language comparison of LTS

In the domain of model-based testing, it is assumed that both the model and the system under test are hidden, with only inputs and outputs being observable. The objective is to verify that the system's inputs yield the expected outputs within the range of inputs and outputs defined by the model. If the outputs are the same, one can be moderately assured that the system is correct. For this, a test set has to be generated. A finite set of tests can be created that can be used for equivalence testing if the maximum number of states in the system is known and the system and model are both minimal and deterministic.

Traditionally, random sampling, i.e. taking random samples that represent the language, is performed to obtain samples that represent the language when comparing LTSs. However, it is unreliable as the samples often do not accurately represent the underlying language. While comparing two sets of sequences is straightforward if they are finite, the structure of an LTS tends to be cyclic, resulting in an infinite number of different sequences. If the LTSs are infinite, an implicit comparison becomes impossible. An alternative approach is introduced in [26] using methods from the model-based testing domain.

W-Method

The W-method [27] systematically generates a finite set of sequences that are representative of a given LTS. The test set ensures

- It reaches every state.
- There are no unexpected outgoing transitions from the states
- The correct state has been reached.

Given a reference LTS A and a model assumed to be a hidden LTS S, the W-Method constructs a set of sequences that should be classified as equivalent by both LTSs. The set $Y \subseteq \Sigma*$ ensures that $(L(A) \cap Y = L(B) \cup Y) \Rightarrow L(A) = L(B)$ and is called the test set of A. The test set is essentially a cross product of three sets [26]:

1. *State cover*: A state cover C is a prefixed-closed set consisting of all sequences of inputs required to reach every state of an LTS A from the start state. $C \subseteq L(A)$ such that $\epsilon \in C$ and for all states $q \in Q\ q_0$ there exists a $c \in C$ such that $\hat{\delta}(q_0, c) = q$
2. *Characterisation set*: For a subset $W \subseteq \Sigma^*$ the states $q1, q2 \in Q$ are called W-distinguishable if $(L(A, q_1) \cap W) \neq (L(A, q_2) \cap W)$. W is a characterisation set of A if any two distinct states of A are W-distinguishable.
3. a set of symbols in Σ

A positive integer k, which estimates the number of extra states that may be required to account for faulty states, is also needed to compute the test set Y. In our case, both models are visible to us so k does not have to be guessed. The test set Y is computed as follows: $Y = C(\{\epsilon\} \cup \Sigma \cup \cdots \cup \Sigma^{k+1})W$:

- C contains sequences that ensure every state is reached.
- $\{\epsilon\} \cup \Sigma \cup \cdots \cup \Sigma^*$ are permutations of Σ up to a length k that try every possible combination of inputs from each state in an attempt to enter unexpected states in the model.
- W ensures the state reached by preceding operations is as intended.

The language of two LTSs A and B can be compared by generating a test set from A using the W-method and measuring the ratio of test sequences classified identically by A and B. k can be found by subtracting the number of states of B from the number of states in A. k can be 0 if B has fewer states.

2.9. Sequence alignment

Sequence alignment is a technique originally developed for computational biology, signal processing and text retrieval. In the bioinformatics and computational biology field, its purpose is to compare and analyze biological sequences, such as DNA, RNA or protein sequences. In the signal processing field, it is mostly used for speech recognition, but also for error correction. Within the domain of text retrieval, its application involves finding relevant information within large text collections. Some examples of sequence alignment are recovering original signals after transmission over noisy channels, finding DNA subsequences after possible mutations, and text searching with typing or spelling errors [28].

Beyond its origins, sequence alignment has found applications in various computer science domains. Sequence alignment algorithms have been adapted to address challenges in text comparison, natural language processing, and virus and intrusion detection. This demonstrates the broad applicability of sequence alignment techniques, showcasing their significance across diverse areas within computer science.

The goal is to identify similarities and differences between two or more sequences by arranging them in a way that maximizes matching characters. This is often done by minimizing the cost to transform the first sequence to the other sequence.

In our application, the possible operations are limited to the ones in Table 2.4.

Operation	Definition
Insertion	Inserting a symbol a at index i of sequence y and index i in sequence x is ϵ
Deletion	Deleting a symbol a at index i of sequence y.
Substitution	Replacing symbol a for symbol b at index i of sequence y, where $a \neq b$.
Transposition	Replacing ab for ba.

Table 2.4: Possible operations in sequence alignment

Various algorithms exist that perform sequence alignment. The *Needleman-Wunsch* and *Smith-Waterman* algorithms are two well-known dynamic programming based sequence alignment algorithms.

Three different types of sequence alignment exist:

1. *Global alignment*: aligns two entire sequences. Best for sequences of roughly the same length and highly similar over their entire length, e.g.:

```
SIMILARITY
PI-LLAR---
```

Figure 2.6: Global alignment of 'similarity' and 'pillar'. From [29]

2. *Local alignment*: aligns parts of two sequences, shorter than the entire sequence, possibly more than one. Suitable for finding regions of similarity within sequences that might differ significantly in other regions, e.g.:

```
MILAR
ILLAR
```

Figure 2.7: Local alignment of 'similarity' and 'pillar'. From [29]

3. *Multiple sequence alignment*: more than two sequences need to be aligned, e.g.:

```
SIMILARITY
PI-LLAR---
--MOLARITY
```

Figure 2.8: Multiple alignment of 'similarity', 'immolarity' and 'pillar'. From [29]

In this thesis it is of importance that the whole sequence of events should fit in the model, thus this thesis will perform *global alignment*: the input sequence will be aligned to the learned DFA.

2.9.1. Distance measure

During sequence alignment, to assess the similarity of two sequences, a cost is calculated for transforming one sequence into the other. For this, a distance measure can be used.

In this thesis, the distance $\delta(x,y)$ between two sequences x and y is the minimal cost of a sequence of operations that transforms string x into string y. The total cost of transforming x into y will be the sum of the cost of individual operations. The operations are a finite set of rules of the form $\delta(z,w)=t$, where *z* and *w* are different strings and *t* is a non-negative real number. Only one operation can be performed on a symbol, so if *z* has been changed into *w*, no further changes can be done on *w*[28]. If for an operation $\delta(w,z)$, the opposite operation $\delta(z,w)$ exists at the same cost, so $\delta(z,w)=\delta(w,z)$, the distance is symmetric. Some of the most common distance measures are:

1. *Levenshtein distance*, also known as *edit distance*, which allows delete, insert and substitute operations in both strings. Generally, the cost of one transform operation is 1. If different costs are assigned, perhaps based on the type of change involved, we speak of the *general edit distance*. An extension of the edit distance is when transpositions are allowed (i.e. a substitution of the form *ab*→*ba*). While the edit distance is simple, it is powerful enough for a wide range of applications [28]. The edit distance is symmetric and the distance will always be less than the length of the longest string: $0 \leq d(x,y) \leq max(|x|,|y|)$.

2. *Hamming distance* which allows substitutions only at the cost of 1 per operation and is also symmetric. If |x| = |y| then the distance is finite and $0 \leq d(x,y) \leq |x|$.

3. *Episode distance* allows only insertions at the cost of 1 per operation. This distance is not symmetric. $d(x,y)$ is either |y| - |x| or ∞.

4. *Longest common subsequence distance* allows only insertions and deletions at the cost of 1 per operation. This distance reflects the longest pairing of symbols that is possible between two strings. In other words, the distance is the number of dissimilar symbols. The distance is symmetric and $0 \leq d(x,y) \leq |x|+|y|$ holds.

2.9.2. Needleman-Wunsch algorithm

The Needleman-Wunsch algorithm [30] is a *global sequence alignment* algorithm originally designed to identify similarities in the amino acid sequences between two proteins. The algorithm uses *dynamic programming*, which employs a divide-and-conquer strategy, breaking down the problem into smaller sub-problems. These sub-problems are solved and their solutions are utilized to solve the original problem. Sequences can differ as follows:

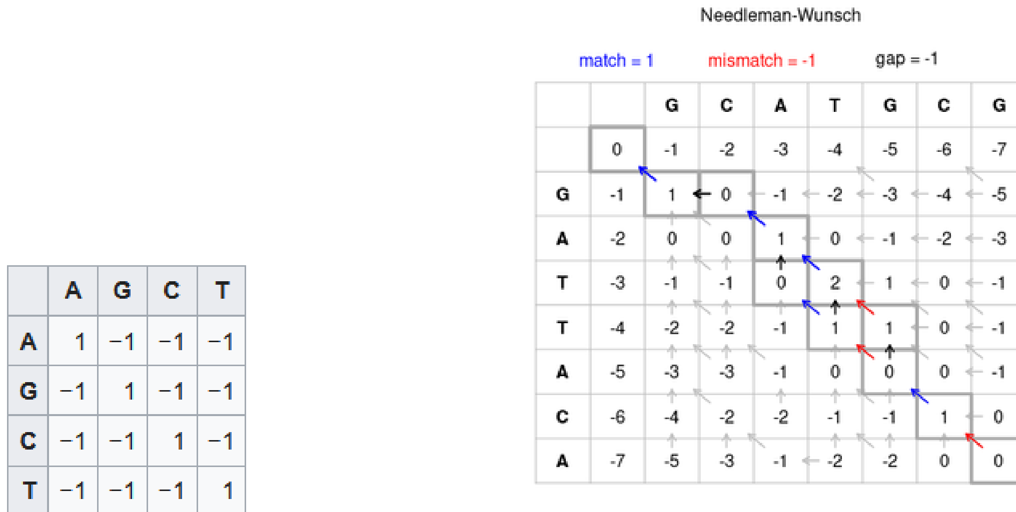
- *match*: two symbols at the same index are the same.
- *mismatch*: two symbols at the same index are different.
- *gap*: sequence B is aligned to A at an index i if the symbol has to be skipped at index i in B.

A *scoring scheme* is needed which assigns a score to the different actions that can be taken.

In the simplest method, the index at (i,j) in the matrix is assigned a value of 1 if the symbol at index i of sequence x and index j of y are the same, otherwise, -1 is assigned, see Figure 2.9a. It is possible to add a *penalty factor* or *gap penalty*, a number subtracted for every gap made, if gaps are allowed. It can be a function prohibiting certain gaps if the penalty exceeds a certain barrier. Conventionally, the penalty for a gap must be several times larger than the penalty for a mismatch. This is relevant in the biomedical field because a gap in the polymer chain can disrupt the entire structure, and in DNA, it can cause a shift in the reading frame [29]. However, if chunks are inserted/deleted, it should be less expensive, as it occurs more often.

It then calculates the *best alignment*; the alignment that maximizes the *alignment score*, which is the total score of the alignment.

The sequences to be aligned are represented in a two-dimensional array. Given sequence x and y of size m and n, the two-dimensional array will be of size $(m+1) \times (n+1)$.



	A	G	C	T
A	1	−1	−1	−1
G	−1	1	−1	−1
C	−1	−1	1	−1
T	−1	−1	−1	1

(a) Simple similarity matrix. From Wikipedia[2]

(b) Needleman-Wunsch pairwise sequence alignment. From Wikipedia[2]

Figure 2.9: Calulating different similarity matrices with Needleman-Wunsch

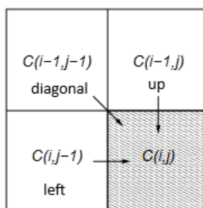


Figure 2.10: Calculation of cell score. From [29]

The algorithm proceeds through the following steps:

1. *Initialize*, see line 3-8 of Algorithm 1, the alignment matrix:

   (a) For x=0 to m assign $j \cdot gapPenalty$ to index M[i][0].

   (b) For y=0 to n assign $i \cdot gapPenalty$ to index M[0][j].

2. *Fill*, see Algorithm 1, in the alignment matrix: at every cell (i, j), pick the best score from *left*: M[i][j-1] ,*up*: M[i-1][j] or *diagonal*: M[i-1][j-1], see Figure 2.10.

3. *Backtrace* the alignment matrix to find the best alignment.

The best match will be the path obtained with backtracing that has the maximum score, starting from the bottom right corner of the matrix $(i = m, j = n)$, to the origin and following the path of the maximum score, see Figure 2.9b. Moving diagonally represents a match, while moving horizontally or vertically represents a GAP.

[2]From Wikipedia: `https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm`

Algorithm 1 Needleman-Wunsch algorithm

Input: size of sequences: m, n
Output: matrix: M
1: **for** $i \leftarrow 0$ to m **do**
2: $M[i][0] \leftarrow$ gapPenalty $\times i$
3: **end for**
4: **for** $j \leftarrow 0$ to n **do**
5: $M[0][j] \leftarrow$ gapPenalty $\times j$
6: **end for**
7: **for** $i \leftarrow$1 to m **do**
8: **for** $j \leftarrow$1 to n **do**
9: *match* $\leftarrow M[i-1][j-1] +$ MatchScore(x_i, y_i)
10: *delete* $\leftarrow M[i-1][j]+$ gapPenalty
11: *insert* $\leftarrow M[i][j-1]+$ gapPenalty
12: $M[i][j] \leftarrow \max($*match,delete,insert*$)$
13: **end for**
14: **end forreturn** M

2.9.3. Smith-Waterman algorithm

The Smith-Waterman algorithm [31] is a *local sequence alignment* algorithm. In contrast to the Needleman–Wunsch algorithm, the Smith–Waterman algorithm resets negative scoring matrix cells to zero. To align two sequences x and y of length m and n a matrix H of size $m \times n$ is created. The algorithm starts with setting H[k][0]=H[0][l] = 0, for $0 \leq k \leq m$ and $0 \leq l \leq n$. Afterwards it fills the matrix using the equation below:

$$H[i][j] = max \begin{cases} H[i-1][j-1] + s(x_i, y_j) \\ max_{k\geq 1} H[i-k][j] - W_k \\ max_{l\geq 1} H[i][j-l] - W_l \\ 0 \end{cases} \tag{2.9}$$

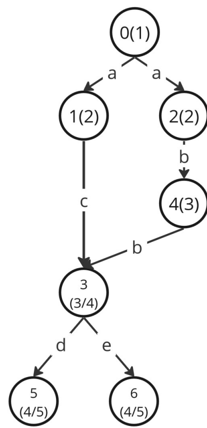
where

- H[i-1][j-1] + $s(x_i, y_j)$ is the score assigned when x_i and y_j are aligned.
- H[i-k][j] - W_k is the score assigned when x_i is at the end of a gap of length k.
- H[i][j-l] - W_l is the score assigned when y_j is at the end of a gap of length k.

The backtrace procedure begins at the matrix cell with the highest score and continues until a cell with a score of zero is reached. This approach identifies the highest scoring local alignment between the sequences.

2.9.4. Tree-Sequence Alignment

The Needleman-Wunsch algorithm was originally created for sequence to sequence alignment. Tsoni [32] has modified the Needleman-Wunsch algorithm to work for sequence to state machine model alignment, where the state machine model is represented as a tree. The goal is to be able to align the sequence with all the sequences that the learned model can generate. First, a level is defined for all nodes. The level of the root node is 1, and after every transition, the level of the destination node is the level of the source node + 1. The following modifications have been made to the Needleman-Wunsch algorithm:



(a) Tree that models the sequences *acd*, *ace*, *abbd*, and *abbe*

symbol	transition	level		a	b	d
			0	-2	-4	-6
a	0->1	1	-2	1	-1	-3
a	0->2	1	-2	1	-1	-3
c	1->3	2	-4	-1	-1	-3
b	2->4	2	-4	-1	2	0
b	4->3	3	-6	-3	0	0
d	3->5	3	-6	-3	-3	-3
e	3->6	3	-6	-3	-3	-3
d	3->5	4	-8	-5	-2	2
e	3->6	4	-8	-5	-2	2

(b) Similarity matrix for the *abd* and the tree, where the match penalty is 1, mismatch penalty is -1 and gap penalty is -2.

Figure 2.11: Tree representation of a FSA and the calculated similarity matrix

- During the initialization, the matrix will be of the size *#transitions in the tree* $\times$ *#length of the sequence*, where every transition is of the form (*source node*, *destination node*), see Figure 2.11a.
- During the filling of the matrix, for every cell the maximum value between left, upper and diagonal neighbour of the cell has to be taken. However, to find the diagonal and upper values the table has to be searched for the cells with *destination node* equal to the *source node* and the level being one less than the current cell, see Figure 2.11b.
- State machine models contain the event as a label on the transition, thus every index of the sequence is compared to an edge in the model.

After the matrix has been initialized and filled, the best alignment can be found with backtracing. Similar to Needleman-Wunsch, backtracing starts at the bottom right corner of the matrix $(i = m, j = n)$, to the origin and following the path of the maximum score. During every move, it has to be checked whether *previous_node* == *source_node*. There are two possible options:

1. If the source node of transition of the current index in the trace is not equivalent to the previous node, it means there is a mismatch edges in the model have been skipped.
2. If the transition is the same as the transition at the previous index, there is an added edge for the current index.

The result of the Tree-Sequence alignment algorithm will return the aligned sequence with a $+$ if an event was added and $-$ if an event was removed. To elucidate an example is presented in Figure 2.11. Given the model in Figure 2.11a, the alignment algorithm will compute the matrix in Figure 2.11b for the sequence 'abd'. Given the matrix, the aligned sequence will be **a,b,d/+|e/+**.

2.10. Evaluation metrics

A fundamental part of any Machine Learning method is evaluating the classification. Classic evaluation metrics include Classification Accuracy, Precision, Recall, Confusion Matrix, Receiver Operating Characteristic (ROC) curve, Area Under the Curve (AUC), F1-score, Mean Absolute Error (MSE), Mean Squared Error (MSE) and Logarithmic loss. These metrics allow for judgements about the system and indicate the quality of a method.

Algorithm 2 Tree-Sequence Alignment matrix computation

Input: edges: E, sequence: S
Output: alignment matrix: M
1: **function** getThreeNeighbouringCells(i, j, *edge, transition*)
2: *upIndex* ← FindUpIndex($edge, i, j$)
3: *left* ← $M[i, j-1]$ + GAP
4: *up* ← $M[upIndex, j]$ + GAP
5: *diagonal* ← $M[upIndex, j-1]$ + MatchScore(*edge.label, transition*)
6: **return** *[left, up, diagonal]*
7: **end function**
8: **function** computeMatrix(matrix, incoming_trace)
9: **for** each *transition* in incoming_trace **do**
10: **for** each *row_key* in matrix **do**
11: *value, position* ← max(getThreeNeighbouringCells())
12: **if** *position* == 0 || *position*==2 **then**
13: **else if** *position==1* **then**
14: **if** *transition* == transition of the *row_key* **then**
15: writeMatrix(*row_key*, *transition*, *value*+MATCH)
16: **else**
17: writeMatrix(*row_key*, *transition*, *value*+MISMATCH)
18: **end if**
19: **end if**
20: **end for**
21: **end for**
22: **end function**

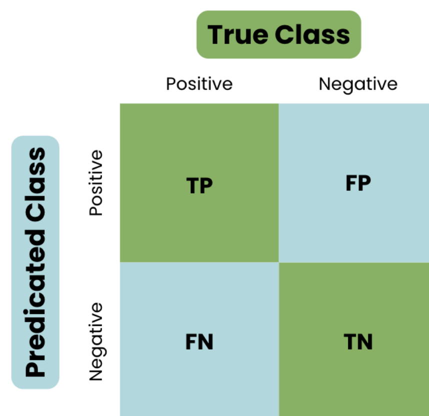


Figure 2.12: Layout of a confusion matrix, from [33]

Before defining the evaluation metrics, the basic building blocks need to be defined. A *true positive (TP)* is an instance that is positive and has been *correctly* predicted to be positive. Similarly, a *true negative (TN)* is an instance that is negative and has been *correctly* predicted to be negative. A *false positive (FP)* is an instance that is negative, but has been *falsely* predicted to be positive. Similarly, a *false negative (FN)* is an instance that is positive and has been *falsely* predicted to be positive.

- **Accuracy** is defined as number of correct predictions out of the total number of predictions.
- **Precision** represents the proportion of positive predictions that were actually correct:

$$Precision = \frac{TP}{TP + FP} \tag{2.10}$$

- **Recall** represents the proportion of actual positives were correctly predicted:

$$Recall = \frac{TP}{TP + FN} \tag{2.11}$$

- **Confusion matrix** is a matrix containing the TP, FP, TN, FN counts for every class, see Figure 2.12. It also works for multi-class classification.
- **ROC curve** shows the TP and FP rates at different thresholds. Typically ,the TP rate is on the y-axis and the FP rate on the x-axis. The top left corner represents the optimal point, where the TP rate is one and FP rate is zero. A threshold defines the value at which the classes are split. For example, if logistic regression has been used, a probability will be returned. If the threshold is 0.6, the values above 0.6 will belong to one class and the values above 0.6 will belong to the other class.
- **AUC curve** measures the area under the entire ROC curve and provides an aggregate measure of the performance across all possible threshold values. It is scale-invariant and classification-threshold-invariant.
- **F1-score** computes the average precision and recall:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} = \frac{TP}{TP + \frac{1}{2(FP+FN)}} \tag{2.12}$$

- **MAE** takes the mean of the differences of each predicted and actual value:

$$MAE = \frac{\sum_{i=1}^{N} |y_i - x_i}{N} \tag{2.13}$$

- **MSE** takes the square of the differences of each predicted and actual value:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2 \tag{2.14}$$

- **Logarithmic loss**, also known as log loss, is an evaluation metric for binary classification. It measures the difference between the actual binary value and predicted probability. The uncertainty of predictions is taken into account by penalizing models more for confident incorrect predictions. The lower the log loss, the better the model performance.

$$LogLoss = -\frac{1}{N} \sum_{i=1}^{N} (y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i)) \tag{2.15}$$

  where N is the number of observations, y_i is the actual binary outcome for the *i*-th observations and p_i is the predicted probability that the *i*-th observation belongs to class 1.

It is important to select an evaluation metrics tailored to the data used as some evaluation metrics can be misleading when there is *class imbalance*. Class imbalance occurs when one class occurs a lot less than the other class(es) causing an disproportionate ratio of instances.

3

Related Work

3.1. Anomaly detection in logs

Anomaly detection in software logs is a well-researched area, using a range of methodologies from machine learning. Various techniques have been applied, including deep learning methods and state machine learning methods.

Fu [34] explores system anomaly detection using FSA, with a focus on performance issues with unstructured log data. A FSA is modeled using the normal system behavior logs. From the unstructured logs, the log messages are extracted without the parameters and clustered into groups based on their similarity, which is measured with an edit distance. A common part in their messages used log key for the group. Afterwards, the unstructured logs are transformed into a structured format, represented by their log key and associated parameters. A sequence of these log keys can then represent the execution path of the program. Once the log data is structured, Fu trains an FSA where each transition corresponds to a log key. For anomaly detection, the log message sequence input is first parsed into a sequence of log keys, which is then checked against the learned FSA. If the sequence does not conform to the FSA, it is raised as a workflow error. The model then proceeds to detect performance anomalies. It compares the execution time of the input sequence to the learned transition times in the FSA. If the execution time exceeds a predefined threshold, it is considered a performance anomaly. Additionally, the model checks for low performance within loop structures using defined thresholds. Fu primarily targets execution anomalies, focusing on workflow issues and performance-related errors. Any sequence that deviates from the learned FSA is flagged as an anomaly, with no insights into the difference in behaviour. Additionally, extreme values in the time taken to transition between states were used to indicate slowdowns in the system.

Anomaly detection has more often been explored using deep learning techniques. One example is *DeepLog* [6], a model proposed for anomaly detection in large volumes of system log data. At its core is a deep neural network employing Long Short-Term Memory (LSTM). Deeplog treats log entries as sequences that follow determined grammar rules and structured patterns, similar to natural language processing. The architecture comprises of three main components: a *log key anomaly detection model*, *a parameter value anomaly detection model*, and a *workflow model* to diagnose detected anomalies. First, each unstructured log entry is parsed into log key and parameter value vector. The parsed sequence is used to train a log key anomaly detection model and system execution workflow models. For each key k, a system performance anomaly detection model is also learned. Anomaly detection

is performed at per log entry level, identifying deviations from the learned model. DeepLog detects performance anomalies by calculating the mean square error (MSE) between prediction and observed parameter value vector. New log entries are first parsed and then go through the log key anomaly detection model, followed by the parameter value anomaly detection model. If an entry is anomalous, the workflow model provides additional information to the expert for in depth diagnosis on the anomaly. A great feature of DeepLog is that it allows the learned model to incrementally adapt and update over time to new log patterns. While DeepLog is able to detect workflow errors and identify which sequences deviate from normal behavior, it gives no insight into the decision-making processes or execution paths in the model. This limitation prevents a direct comparison with normal behavior, as normal behavior is stored implicitly within the LSTM model.

3.2. Log-based behavioural differencing

An extension of anomaly detection in software logs involves gaining insight into the differences between logs. Several tools have been developed for behavioral differencing.

Goldstein [35] developed a tool that detects anomalies and visualizes behavioral differences between normal and abnormal executions by learning a behavioral model with FSA from both types of behavior. It is similar to Fu [34], with the addition of difference computation and visualization. Additionally, the tool does not limit transitions that do not fit the normal model. The tool operates in four steps:

1. Log normalization by extracting log keys and transforming the log data into normalized or structured log data.

2. Behavioral model extraction in the form of a FSA using the kTails [36] algorithm.

3. Difference computation of the two models by comparison of FSA models with a difference model. The difference model has all nodes from both models: common nodes are nodes in both models, nodes only in the second model are depicted as *added* nodes and nodes that are only in the first model are depicted as *removed* nodes.

4. Visualization of the extracted differences, automatically pointing out: *execution flow outliers* and *transition time performance degradations* with bold and dashed lines and colors.

The tool is able to analyze millions of lines of logs, reducing the effort required by experts during log data investigation. However, the tool's effectiveness is limited to models with a relatively small number of nodes, not more than a couple of dozens. Otherwise the tool may generate all possible differences, leading to long run times. It is possible to limit the number differences that the tool adds to the visualization, but this limits the exhaustiveness and accuracy of the tool. Furthermore, the tool approximates the logs, causing the comparison to contain more differences that may not necessarily be evident in the logs. Lastly, the tool only allows the comparison of two models.

De Knop optimized the visualization tool created by Goldstein [35] for log-based behavioral differencing, specifically addressing the challenges of visualizing logs before and after software refactoring. Graphs produced during refactoring tend to be very linear with long chains of nodes with few changes and transition probabilities of 1. This linearity makes it difficult to distinguish meaningful patterns and differences. Additionally, changes often occur in clusters. To address these issues, De Knop introduced an enhancement to the visualization tool with a merge algorithm to be applied after the original differencing algorithm by Goldstein [35]. The merge algorithm ensures no key insights are lost. By collapsing long chains into single nodes, the algorithm significantly improves readability. De Knop manages to

reduce the output size of the original algorithm while not hiding important information, allowing for more accessible analysis.

Similar to Goldstein [35], Amar [5] presents two algorithms: *2KDiff* and *nKDiff*, both based on the classic kTails algorithms. 2KDiff highlights the specific traces that show the differences between two logs, while nKDiff highlights no specific traces but detects differences across many logs at once. *2KDiff* takes two logs as input and creates a k-FSM for each log, where *k* is a positive integer. It computes the set of k-sequences in each log and identifies the k-differences, which are the k-sequences unique to each log. Using a greedy approach, 2KDiff selects a minimal set of traces in each log that covers all k-differences. These traces are then replayed over the k-FSMs, with the transitions corresponding to k-differences represented as highlighted paths on the output FSM. *nKDiff* takes a number of input logs $L_1, \dots, L_n$, labels each log, and computes the alphabet of all input logs. It starts with the creation of a labeled FSM (LFSM), where each trace's sequence of events is represented as a linear sub-FSM. Each of the sub-FSMs is labeled with an index corresponding to the log it originated from. All states in the LFSM are merged into equivalence classes, based on the states' futures, which define the states of the output k-DiffLFSM. Differences are highlighted as transitions that do not appear in any overlapping sub-FSMs. It outputs a single labeled FSM representing the differences. 2Kdiff and nKDiff help experts with in comparing execution logs. However, the approach is limited to k-differences and reports all differences with equal importance, highlighting entire traces even if only a small part differs.

4 Methodology

4.1. Data Exploration

This section introduces the datasets used for training the models and evaluating the sequence alignment against the learned models. In practice, software log data is raw, unstructured, and often unlabeled. The data used in this work will reflect this nature. The training set consists of *positive data*, which represents normal traces and the expected behavior of the system. During the evaluation of the sequence alignment, both positive (normal) and negative (anomalous) data will be aligned to the model and evaluated as normal or anomalous.

4.1.1. Data preprocessing

The data is processed to be compatible with Flexfringe. The following preprocessing steps are required to transform the data into Abbadingo format:

1. Parsing the logs by extracting event templates from unstructured logs and converting raw log messages into a sequence of structured events.
2. Reconstructing the traces.
3. Count number of traces and alphabet size.

To parse the log events into distinct keys, Logparser[1] by Logpai is used. Logparser extracts event templates from unstructured logs and converts the raw log events into a sequence of structured events. This process is also known as message template extraction, log key extraction, or message clustering.

The traces are then reconstructed by grouping execution sequences of related events. For example, the events for opening, reading or modifying and closing a file would constitute an event trace for that file. These events can often be grouped by some key identifier that distinguishes different objects. All events with the same identifier constitute a trace.

4.1.2. Hadoop Distributed File System

he Hadoop Distributed File System (HDFS) dataset[2] collected by [3], is a distributed file system designed to run on commodity/low-cost hardware. It is a popular system and has thus been widely studied

[1] `https://github.com/logpai/logparser`

[2] HDFS data: `https://github.com/Thijsvanede/DeepLog/tree/master/examples/data`

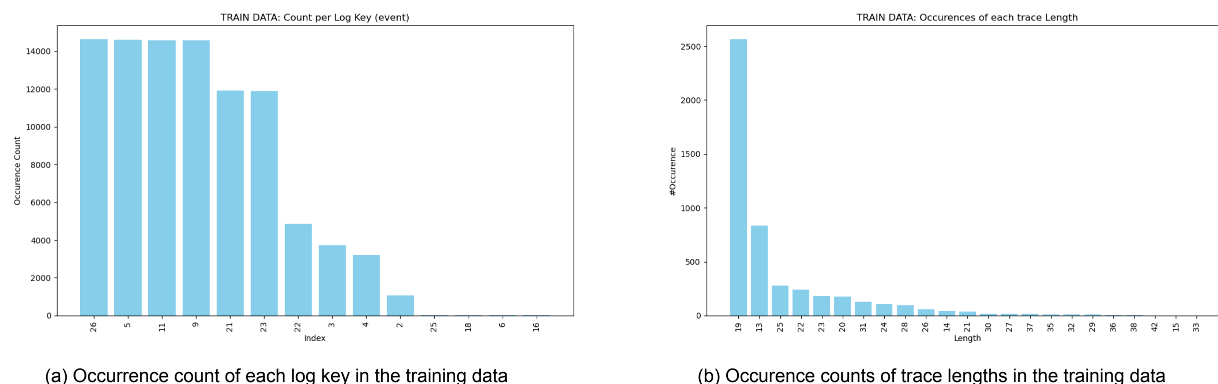
in literature. The dataset yields 24 million lines of logs from a Hadoop cluster running on over 200 EC2 nodes. The data is split into a training set (4.855 lines), a test set with positive traces (553.366 lines) and a test set with abnormal traces (16.838 lines), where each line is one log trace and every number represents an event.

The HDFS dataset[3] used has been processed such that the unstructured log entries are parsed into log keys and grouped by their unique identifiers. Each line in the dataset represents the execution sequence of a process. For details on the data collection and parsing, one may refer to [37] and [38].

While the dataset is no longer unstructured, it is not in Abbadingo format yet. Step 3 from the preprocessing steps mentioned in Section 4.1.1 needs to be executed, such that the dataset starts with a line containing the *number of traces* and *alphabet size*, and each trace starts with the *type* and *length of trace*. After preprocessing, the dataset should resemble the format shown in Figure 2.5.

Data exploration

The training data consists of various log keys (events) that occur a varying number of times within a trace. A log key may occur frequently or not at all. The occurrences of different log keys in the entire training dataset are shown in Figure 4.1a. The most common log keys are 26, 5, 11, 9, 21, and 23, while the rarest log keys are 25, 18, 6, and 16. The lengths of the traces are most often 19, as shown in Figure 4.1b



(a) Occurrence count of each log key in the training data

(b) Occurence counts of trace lengths in the training data

Figure 4.1: Comparison of occurrence counts and trace lengths in the training data

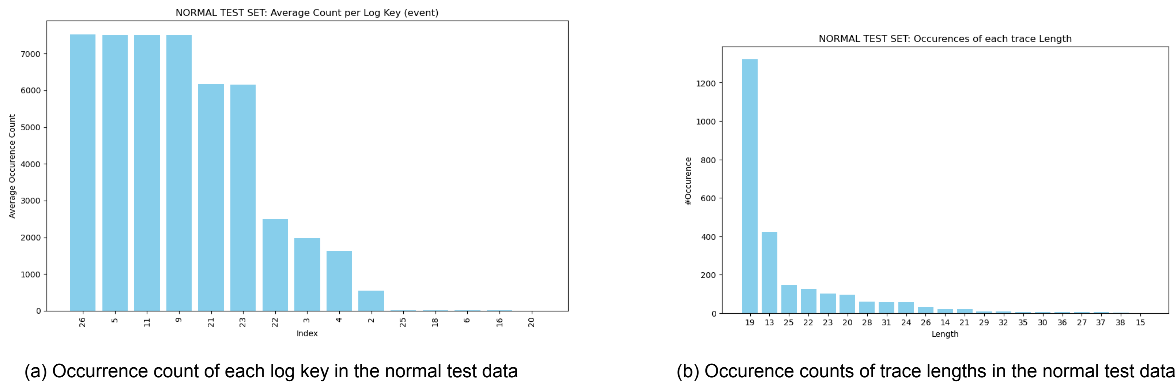
When manually inspecting the training data, several characteristics are noted:

- Most traces begin with a sequence of 5's, 22's or a combination of 5's and 22's.
- Are followed by alternating 11 and 9's or a sequence of 26's.
- Can be followed by 2's, 3's and 4's, which can also occur interchangeably.
- Most often end with a sequence of 23's and a sequence of 3 times 21's.
- Sometimes end with 26.

From Figure 4.2, it is apparent that the log key counts and trace lengths in the normal test data are similar to the log key counts and trace of the training data, see Figure 4.1.

In the abnormal test data, certain log keys appear that are not present in the positive examples (training data and normal test data), as shown in Figure 4.3a. However, the top six most frequent log keys are the same as those in the positive examples. The most significant difference between the abnormal and normal data lies in the trace lengths. Figure 4.3b clearly shows that many traces have lengths of 2 and 3, which never occur in the positive examples.

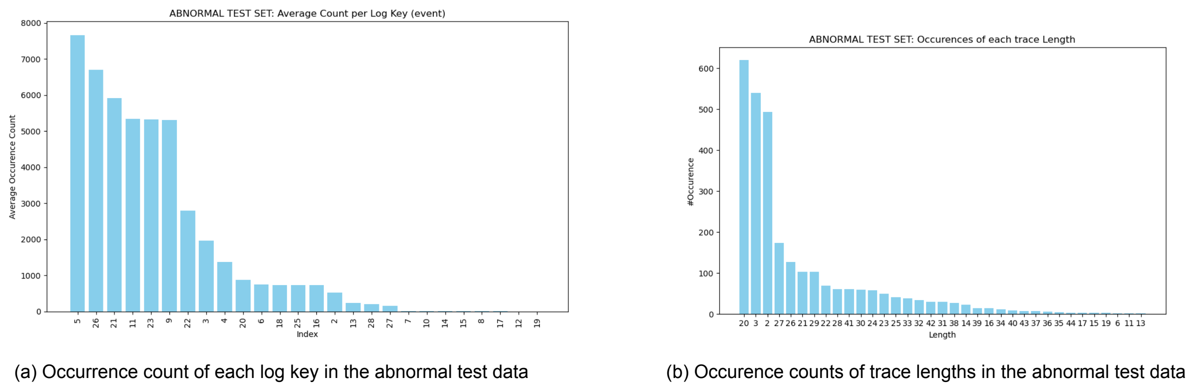
[3] https://github.com/Thijsvanede/DeepLog/tree/master/examples/data



(a) Occurrence count of each log key in the normal test data

(b) Occurence counts of trace lengths in the normal test data

Figure 4.2: Comparison of occurrence counts and trace lengths in normal test data



(a) Occurrence count of each log key in the abnormal test data

(b) Occurence counts of trace lengths in the abnormal test data

Figure 4.3: Comparison of occurrence counts and trace lengths in abnormal test data

Due to the large size of the normal test data, the runtime for running the sequence alignment on the dataset exceeds half an hour. To address this, the normal test data are split into sets of 20,000 lines each, resulting in 28 subsets of the normal data.

4.2. Modelling data with FSA

To model the behavior of software logs using a FSA, more specifically a DFA, the state machine learning tool Flexfringe is employed.

The HDFS dataset is large and sparse, containing events that may occur hundreds of times as well as events that may occur only once. Consequently, the FSA learned from this dataset will be large and potentially difficult to interpret. To make the model more concise and easier to visualize, two parameters will be set: `-sinkson` and `-sinkcount`. It is important to select an appropriate sink count to ensure that the model does not overfit or underfit the data. This section will discuss the heuristics and settings selected in Flexfringe for the HDFS dataset.

4.2.1. Heuristic selection

As mentioned previously, Flexfringe has implemented the following evaluation functions: Alergia, Likelihoodratio, MDI and AIC, EDSM, and overlap driven.

HDFS dataset

The HDFS dataset is very large, which can cause many of the tests performed by Alergia on each pair of states to fail, resulting in a larger and sub-optimal DFA. Likelhoodratio aims to solve this issue by

computing a single test for the entire merge procedure. MDI uses the Kullback-Leibler divergence to ensure that the distribution of the original data and the learned model are similar; however, it is very computationally expensive. In fact, running Flexfringe with the MDI evaluation function on the training data was halted after 36 hours. AIC focuses on balancing the model complexity and the goodness of fit, considering both the likelihood and the number of parameters. EDSM requires both positive and negative data, whereas the HDFS dataset only contains positive data, resulitng in a model with only one state, see Appendinx A.1.7. Overlap Driven's approach of favoring states with overlapping symbols and its success in the StaMInA competition suggest it could be highly effective for the HDFS dataset, which may have a large alphabet and sparse data.

Based on these factors, Alergia, MDI, and EDSM are not suitable for this thesis. While Likelhoodratio and overlap driven are also viable options, AIC is a widely used heuristic and is thus selected as evaluation function.

The following command is used to model the DFA with Flexfringe:

```
.\out\build\x64-Release\flexfringe.exe ..\hdfs_train_abbadingo.txt --
    heuristic-name aic --data-name aic_data --sinkson=1 --sinkcount=n
```

The commands are specified in the following order: `build`, `train data file`, `heuristic-name`, `data-name`, toggle sinks, `sinkcount`, where n>0.

4.2.2. Sink count

The goal is to select a model that represents the data in an understandable DFA, avoiding many parallel lines and fork constructs. Different DFAs will be learned with sink counts of 5, 10, 50, 100 and 300. The model with the largest sink count that still captures the behavior accurately and reduces the signs of parallelism will be used for the subsequent experiments.

4.3. Sequence alignment

After the DFA is learned, both normal and anomalous test cases need verification within against the model. If a sequence does not conform to the model, sequence alignment is performed to attempt to align it to the model. The sequence alignment algorithm is be integrated into the Flexfringe codebase.

The Tree-Sequence Alignment algorithm presented in Section 2.9.4 aligns a sequence to a state machine model represented as a tree. It uses the levels of nodes to match *source nodes* with *destination nodes*, where the source node is one level lower than the destination node, in order to find the previous state in the model.

The most significant modification between the sequence alignment implemented in this thesis and the Tree-Sequence Alignment algorithm occurs during the matrix calculation. In our approach, the alignment process allows skipping to any other state in the model when a mismatch is encountered, regardless of the level of the states. As a result, the requirement for the nodes to be at a specific level is discarded. Instead of restricting nodes to those that are 1) source nodes of the current transition and 2) one level below the current transition level, we now apply only the first criterion during the diagonal and upper value calculations. This adjustment expands the pool of nodes considered, providing more opportunities to skip and align transitions.

The alignment that is implemented in this thesis consists of three steps:

1. **Initialize the matrix** of size *#edges in the automaton* × *#length of the sequence*.

   The matrix contains every (*source_node*, *transition_node*) pair of the automaton at each index of the sequence.

Look up in the matrix is performed on *destination_node* of the transition, then the *symbol_index* in the trace and lastly on the *source_node*

2. **Fill the matrix**, see Algorithm 2.

   At every index of the sequence, starting from the first, the symbol at that index is checked against all edges in the entire automaton using breath first search starting from the root of the automaton, filling each cell in the matrix.

   Breath first search is chosen to ensure the diagonal, left and upper cells have already been filled in from the previous calculation, avoiding empty cells, causing errors in the cell calculation of the current state.

   As in the Needleman-Wunsch algorithm, the maximal value of the diagonal cell value plus either match or mismatch score, left cell value plus gap penalty, and upper cell value plus gap penalty is selected.

   However, multiple diagonal and upper cell values are possible, each of which are taken into consideration and stored in the matrix.

3. **Backtrace to acquire the best alignment.** Starting at the last column of the matrix, which is the last index in the sequence, select the transitions with the highest score. Proceed until the first index of the sequence. This can result in more than one possible alignment. Therefore, at every transition it has to be checked whether *source_node* equals *transition_node*. For the sake of this thesis, we only output one possible alignment.

The algorithm will return the aligned sequence and an *alignment score*. A '+' represents an added edge and a '−' represents a skipped symbol in the trace or a skipped edge in the model. It is possible that there are more ways to align a sequence if for an index in the trace, there are several transitions with the same score, other possible transitions are indicated with a '|'.

Three types of scoring are experimented with: static, linear and dynamic scoring. To allow the user to easily select one of these scoring types with the Needleman-Wunsch sequence alignment, the following parameters are implemented:

1. `--predictalignnw`, toggle for the implemented Needleman-Wunsch sequence alignment.

2. `--nwscoring`, scoring mechanism for predict align with Needleman-Wunsch sequence alignment, options: 'static', 'linear' or 'dynamic'. Default=static.

To run the prediction, the following command is used:

```
.\out\build\x64-Debug\flexfringe.exe ..\hdfs_test_normal.txt --ini ini\aic
    .ini --mode=predict --predictalign=1 --predictalignnw=1 --nwscoring=
    linear --aptafile=..\\hdfs_train_abbadingo.txt.ff.final.json
```

The commands are specified in the following order:

1. `build`, path to the build file

2. path to the test data file

3. either an .ini file or the `heuristic-name` and `data-name`.

4. `predictalign`, toggle for original predictalign already implemented in Flexfringe.

5. `predictalignnw`, toggle for prediction with Needleman-Wunsch sequence alignment.

6. `nwscoring`, set scoring mechanism for predict align with Needleman-Wunsch.
7. `aptafile`, file path containing learned model by Flexfringe

It is required that the evaluation function used for the learned model from the APTA file matches the selected evaluation function, in this case it will be the AIC evaluation function.

4.3.1. Evaluation

To evaluate the performance of an alignment produced by the alignment algorithm, the final score of the aligned sequence, derived from the alignment matrix, is normalized to ensure that the sequence length does not affect the evaluation. When referring to the alignment score, it always denotes the normalized alignment score. The score can be calculated as:

$$\text{score} = \frac{\text{alignment score} - (\text{length of trace} \times \text{\#mismatches})}{\text{length of trace} \times \text{\#matches} - \text{length of trace} \times \text{\#mismatches}} \tag{4.1}$$

Sequences that contain event keys that are *not in the model alphabet* and sequences that *end in a non-final state*, automatically are assigned an alignment score of 0. Sequences that have a score of 1 fit perfectly in the automaton and required no alignment.

4.3.2. Classification

The sequence alignment algorithm returns the aligned sequences along with a normalized alignment score ranging from 0 to 1. To classify the aligned sequences, a threshold needs to be set, which determines whether the traces are labeled as normal (1) or anomalous (0).

The thresholds are selected as follows:

- The first threshold is the **minimum** alignment score achieved in the training set, that is larger than 0.
- The second threshold is the **first quartile** of the alignment scores of the training set.

5 Experiments

This section presents three experiments that aim to answer the research questions 1, 2, 3, and 4.

5.1. DFA model

To select the appropriate sink count, DFA models with different sink counts were learned. The DFA model without any sinks had a size of 3653 nodes. This size makes the model too large and difficult to interpret visually, also rendering it impossible to include the entire model in the Appendix. To reduce the model in size, models with different sink counts were learned resulting in DFAs of different sizes, see Figure 5.1.

Table 5.1: DFA sizes with different sink counts

Model	Sinks	Size (in nodes)
AIC	none	3653
AIC	5	444
AIC	10	189
AIC	20	94
AIC	50	75
AIC	100	55
AIC	300	37

The most significant improvement in the model's ability to learn the unique patterns occurs when the sink count is set to 50, see Appendix A. The model with sinkcount=300 captures the overall behavior and is much more readable and simple compared to the models with a sink count of 50 and less. For details, refer to Appendix A.1.1 and A.1.5. Therefore, Experiment 1 is performed on three models with sink count of 50, 100 and 300. Experiments 2 and 3 will only run on the model with sinkcount=300.

5.2. Experiment 1: Scoring

This section aims to answer research question 1: *"How effective is sequence alignment on a compact model of behavior in detecting flows that deviate from expected behavior?"*.

For Experiment 1, three models are selected with sink count 50, 100 and 300, see Appendix A;

5.2.1. Static scoring

In the first experiment, sequence alignment uses static scoring. Where the match, mismatch and gap penalties are static values. To find the optimal scoring, arbitrary values between a range are tried on the train dataset. For the match score, values between 1 and 10 are considered. For the gap penalty, values between -1 and -7 are considered. Lastly, for the mismatch penalty values between -1 and -12 are considered. It should be noted that the match score is always be positive as matches should be rewarded. The gap penalty is smaller than the mismatch penalty as a gap equals a jump and we want to motivate the model to perform jumps such that an alignment can be found. The best performing combination is chosen for static scoring, and also used for the subsequent experiments.

5.2.2. Linear scoring

The linear scoring takes into consideration how large the current gap in the sequence already is. The gap penalty scales with the length of the existing gap. The motivation behind this is that a larger gap generally indicates a larger deviation from normal behavior. Therefore, a larger gap should be penalized more. To illustrate this, consider the sequence *a**ee**e*, where the bold part represents a gap. During the alignment of this sequence to the automaton shown in Figure 2.11a, the gap penalty will be at the first gap -1, and $2 * -1 = -2$.

5.2.3. Dynamic scoring

Dynamic scoring penalizes jumps to parts in the automaton that are further away more than jumps to parts of the automaton that are closer. This is because a jump to a more distant part of the automaton represents a greater difference from the state of the previous index. It is important to reflect this in the gap penalty. To take the distance into consideration the function *merged_apta_distance* from Flexfringe is used. This distance metric is based on the distance between the two states under consideration and their mutual ancestor node. The gap score is calculated by multiplying the distance with the gap penalty.

5.2.4. Results

Several match, mismatch and gap penalty scores were tested, as shown in 5.2. These values were selected at random within the following ranges: match scores between 1 and 10, mismatch scores between -2 and -12, and gap scores between -1 and -7. The best scoring combination was a match score of 2, mismatch penalty of -10 and gap penalty of -1.

The average alignment scores for each model, with sink counts 50, 100 and 300 and each scoring type, are presented in Table 5.3 for the training set, in Table 5.4 for the normal test set, and in Table 5.5 for the abnormal test set. The scores for the normal data are consistently above 0.9 across most scoring types, except for dynamic scoring on the model with sinkcount=300. Performance on the train and normal test set appears to be similar across all scoring types. In contrast, the scores for the abnormal data are consistently below 0.16, highlighting a distinct difference between normal and abnormal data.

As the sink count increases, the average alignment scores decrease across all models with static and linear scoring on all datasets. This outcome is expected, as traces will fit the model less well. Dynamic scoring on the model with a sinkcount=300 performs worse compared to other models and scoring types for the training set and normal test set. A higher sink count can reduce the closer states available for the algorithm to jump to, resulting in jumps to more further parts in the automaton. This increases the distance of jump operation, resulting in a higher gap penalty and lower alignment score. Furthermore, the algorithm is more inclined to jump to closer states, even if jumping to a further state could result in the sequence ending in a final state.

Table 5.6 presents the calculated thresholds for every model, derived from the minimum and first

Table 5.2: Different scoring tested on training data and a model learned with AIC and sinks=300

match	mismatch	gap penalty	alignment score
1	-2	-1	0.920656
2	-3	-2	0.911173
2	-4	-3	0.904356
2	-9	-1	0.969516
2	-10	-1	**0.972057**
2	-10	-3	0.949932
2	-10	-6	0.916325
3	-10	-1	0.966415
4	-11	-1	0.969516
4	-12	-2	0.955291
5	-10	-7	0.908949
6	-10	-7	0.908309
10	-6	-4	0.905693

Models	Static scoring	Linear scoring	Dynamic scoring
AIC, sinks=50	0.987714	0.985216	0.920497
AIC, sinks=100	0.985193	0.982254	0.958306
AIC, sinks=300	0.95499	0.932214	0.794599

Table 5.3: Average alignment scores with different scoring for three AIC models on the training set

Models	Static scoring	Linear scoring	Dynamic scoring
AIC, sinks=50	0.987259	0.984359	0.919734
AIC, sinks=100	0.983361	0.979874	0.955431
AIC, sinks=300	0.953033	0.929921	0.791148

Table 5.4: Average alignment scores with different scoring for three AIC models on the normal test set

Models	Static scoring	Linear scoring	Dynamic scoring
AIC, sinks=50	0.150035	0.139909	0.111026
AIC, sinks=100	0.149369	0.139693	0.124971
AIC, sinks=300	0.147658	0.134126	0.136654

Table 5.5: Average alignment scores with different scoring for three AIC models on abnormal test set

quartile scores of the training data. Using these thresholds, the confusion matrices were plotted, see Appendix B.3, and the F1-scores were calculated for the Needleman-Wunsch (NW) alignment on the training, normal and abnormal test sets. Additionally, the F1-scores for predict without alignment, `predictalign` from Flexfringe were computed for comparison, see Table 5.7. For the corresponding confusion matrices and ROC curves, refer to Appendix B.

From the F1-scores, it is apparent that the minimum alignment score as threshold always outperforms first quartile score as threshold. Moreover, when comparing the NW-alignment with predict and `predictalign` from Flexfringe, `predictalign` shows the best performance for models with a lower sink count (<300). However, when the sink count increases from 100 to 300, the F1-score from

Models	Scoring	Min value (>0)	First quartile
AIC, sinks=50	Static	0.842342	1.0
AIC, sinks=50	Linear	0.691441	1.0
AIC, sinks=50	Dynamic	0.891892	1.0
AIC, sinks=100	Static	0.842342	0.9875
AIC, sinks=100	Linear	0.691441	0.9875
AIC, sinks=100	Dynamic	0.8875	0.991667
AIC, sinks=300	Static	0.842342	0.947368
AIC, sinks=300	Linear	0.361111	0.929825
AIC, sinks=300	Dynamic	0.878378	0.95614

Table 5.6: Average alignment scores with different scoring for three AIC models on abnormal test set

`predictalign` drops significantly and the NW-alignment with both static and linear scoring perform better.

Upon examining the confusion matrices, it is apparent that with NW-alignment, using the minimum score as a threshold, most errors are FP, with few FN. In contrast, when using the first quartile score as a threshold for NW-alignment, the majority of errors are FN with very few FP. The `predictalign` method shows a similar pattern to NW-alignment with the first quartile threshold, but with less errors overall, exhibiting mostly FN and very few FP.

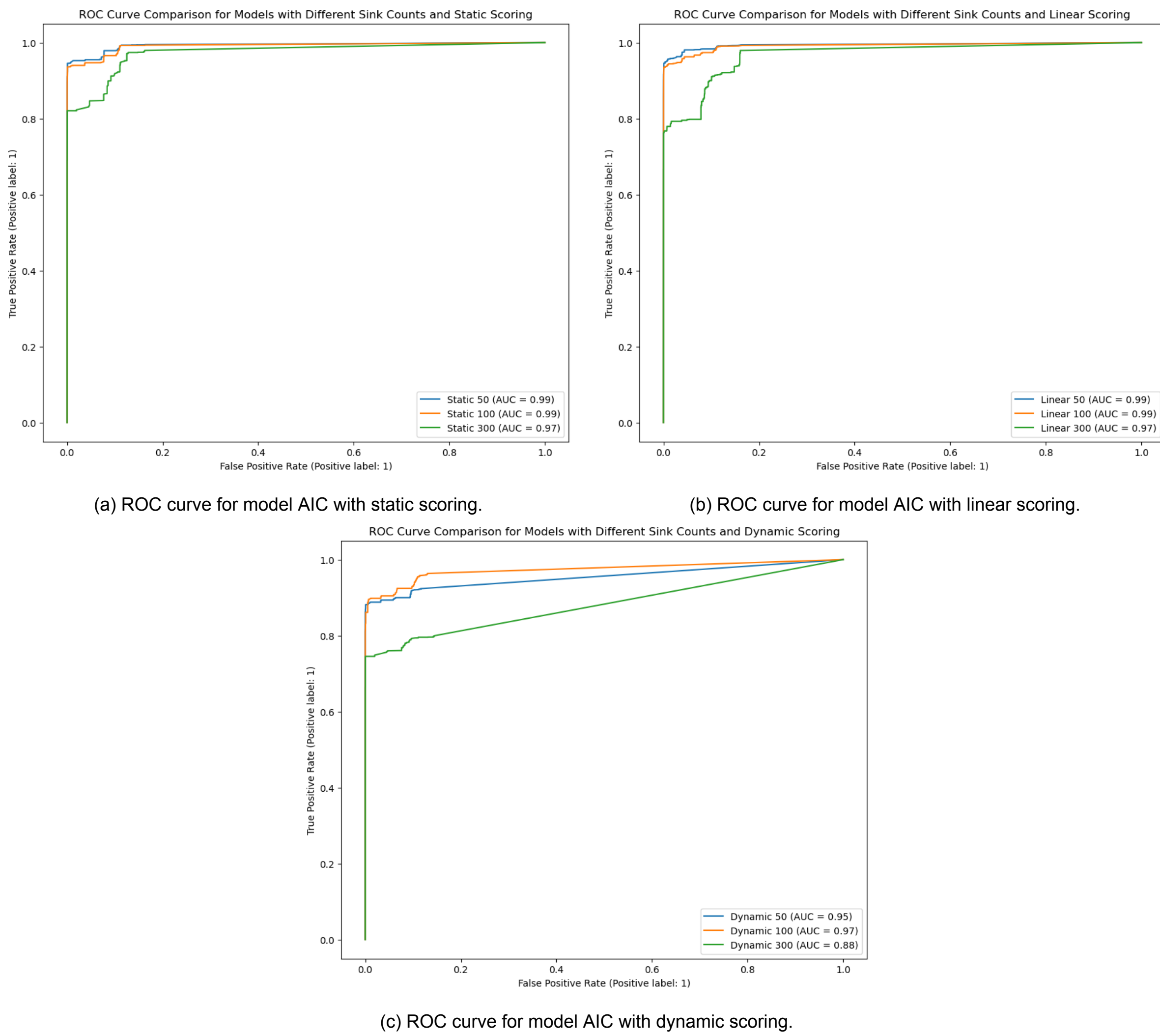
Models	Predict	Flexfringe PredictAlign	NW PredictAlign (Static)	NW PredictAlign (Linear)	NW PredictAlign (Dynamic)
AIC, Sinks=50, min	0.9288	**0.9830**	0.9449	0.9446	0.9576
AIC, Sinks=50, fq	0.9288	**0.9830**	0.8866	0.8866	0.8877
AIC, Sinks=100, min	0.9086	**0.9664**	0.9442	0.9440	0.9536
AIC, Sinks=100, fq	0.9086	**0.9664**	0.8639	0.8631	0.8511
AIC, Sinks=300, min	0.8959	0.9073	0.9373	**0.9378**	0.8427
AIC, Sinks=300, fq	0.8959	**0.9073**	0.8688	0.8624	0.8497

Table 5.7: F1-scores of no alignment, Flexfringe predictalign and NW-predictalign.

Based on the ROC curve and AUC score for all scoring methods, see Figure 5.1, sequence alignment performs similarly for models using static and linear scoring. Models with sinkcount=300 performs slightly worse across all scoring types and the performance of dynamic scoring is worse across different sink counts compared to static and linear scoring.

The ROC curves from predict, `predictalign` from Flexfringe and NW-alignment for sinkcount=300, see Figure 5.2, support the earlier result that NW-alignment outperforms prediction and `predictalign` when the sink count is 300. A model with sinkcount=300 with NW-alignment scores identical AUC scores with a `predictalign` with sink count 5, 10, 100 with `predictalign` and always scores better than predict, see Appendix B.4. Showing its improvement on regular prediction and its effectiveness in behavioral differencing when sinks are turned on.

To demonstrate the impact of sequence alignment, we will analyze the specific trace *”5 5 5 22 11 9 11 9 11 9 26 26 26 3 4 3 3 4 3 23 23 23 21 21 21”* from the normal test set, see Table 5.8 for the aligned state sequence. This trace fits the model with sinkcount=50, but for the models with a higher sink count, the trace needs to be aligned. To review the aligned state sequence, refer to the models

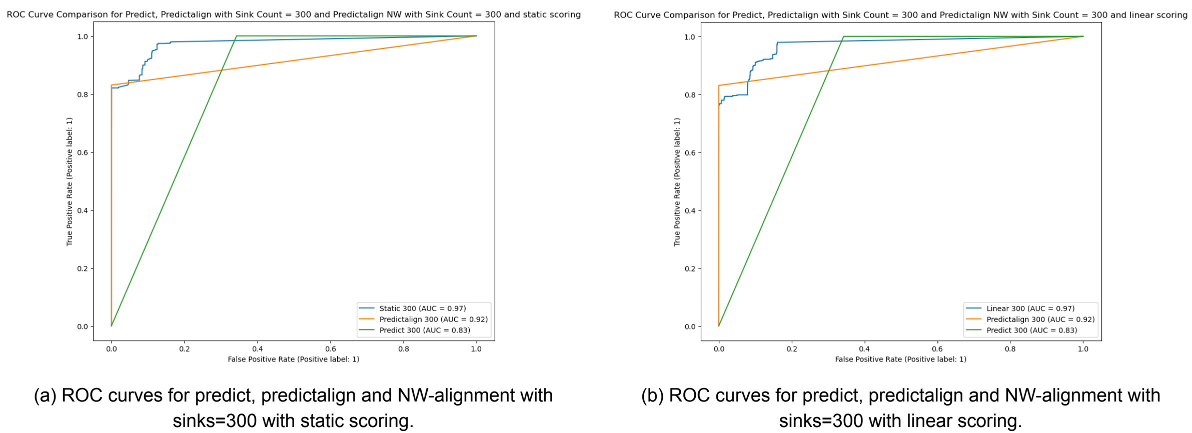


(a) ROC curve for model AIC with static scoring.

(b) ROC curve for model AIC with linear scoring.

(c) ROC curve for model AIC with dynamic scoring.

Figure 5.1: ROC curves for model AIC with different sink counts and scoring methods.



(a) ROC curves for predict, predictalign and NW-alignment with sinks=300 with static scoring.

(b) ROC curves for predict, predictalign and NW-alignment with sinks=300 with linear scoring.

Figure 5.2: ROC curves for model AIC with different sink counts and scoring methods.

in Appendix A. The aligned sequences are given in Table 5.9, where symbols followed by /- indicate symbols that were skipped in the automaton or skipped in the sequence, and those with /+ indicate jumps made by the model to read the symbol.

Model	States	Alignment score
Sinks=50	[1,4,10,18,29,50,88,147,232,345,482,645,820,1004,1322,1727,2189, 2746,3365,1002,1314,1715,1715,1715,1715]	1
Sinks=100	[1,4,10,18,29,50,88,147,232,345,482,645,820,1004,1727/+,1321/+ ,1321/-,1727,1727/-,1002/+,1314,1715,1715,1715,1715]	0.97
Sinks=300	[1,4,10,18,29,50,88,147,232,345,482,645,820,820/-,820/-,820/-,820/- ,820/-,820/-,1002,1314,1715,1715,1715,1715]	0.94

Table 5.8: States of aligned sequence of trace: *"5 5 5 22 11 9 11 9 11 9 26 26 26 3 4 3 3 4 3 23 23 23 21 21 21"*

Model	States
Sinks=50	5 5 5 22 11 9 11 9 11 9 26 26 26 3 4 3 3 4 3 23 23 23 21 21 21
Sinks=100	5 5 5 22 11 9 11 9 11 9 26 26 26 3 4/+ 3/+ 3/+ 3/- 4 4/- 23/+ 23 23 21 21 21
Sinks=300	5 5 5 22 11 9 11 9 11 9 26 26 26 26/- 26/- 26/- 26/- 26/- 26/- 23 23 23 21 21 21

Table 5.9: States of aligned sequence of trace: *"5 5 5 22 11 9 11 9 11 9 26 26 26 3 4 3 3 4 3 23 23 23 21 21 21"*

5.3. Experiment 2: Parallelism

This experiments aims to answer research question 2: *"What are the characteristics of parallel processes in FSA and what rules can be added to the sequence alignment to effectively recognize these?"* and 3: *"How effective are these rules in combination with sequence alignment on recognizing parallel processes in a compact model of behavior?"*.

As explained in Section 2, parallelism causes log entries to be out of order or have different orders every time. Some examples of parallelism in logs are:

Example 1: Interleaved events

This occurs when events happen in an interleaved manner, for example:

1. 5 22 5 5 **11 9 11 9 11 9** 26 26 26 23 23 23 21 21 21
2. 5 5 5 22 **9 11 9 11 9** 26 26 **11** 26 23 23 23 21 21 21

The bold part represents a sequence of interleaved events.

It is also possible that sub-sequences occur in an interleaved manner. For example:

1. 22 5 5 5 11 9 **(26 26 26) (11 9 11 9)** 2 4 4 3 23 23 23 21 21 21
2. 5 22 5 5 11 9 **(11 9 11 9) (26 26 26)** 2 23 23 23 21 21 21

Where the sub-sequences are indicated with brackets.

Example 2: Swapped events

This occurs when when traces are almost identical but some events have been swapped, for example:

1. 5 5 5 22 11 9 11 9 11 9 26 26 26 **3 4** 3 4 3 3 23 23 23 21 21 21
2. 5 22 5 5 11 9 11 9 11 9 26 26 26 **4 3** 4 23 23 23 21 21 21

The bold part indicates the swapped events.

5.3.1. Ignore-skip rule

To enhance the detection of parallelism, a rule allowing the recurrence of a skipped event is implemented. This means that the algorithm is allowed to temporarily ignore one event in order to detect potential parallel structures, and when that event reappears later in the sequence, it can be processed without any cost. However, only events that occurred within a certain distance of the current event should be allowed to be skipped. This ensures that only recent events are considered relevant. This prevents an event ignored at the beginning of a sequence from being allowed to be skipped at the end of a long sequence. In this case, an arbitrary distance of 3 is chosen. This approach introduces an additional step during the cost calculation in each matrix cell, as it requires checking the skipped symbols from the previous three indices against the current symbol.

5.3.2. Results

From Table 5.10, it appears alignment scores show only a slight increase. The F1-score stays the same with 0.9373 for the model with sinkcount=300 and static scoring and 0.9373 for the same model with ignore-skip rule when threshold is the minimum score. However, the F1-score decreases to 0.8688 for the same model with ignore-skip rule when threshold is the first quartile score. The distribution of the confusion matrices, see Figure 5.4, is similar as without the ignore-skip rule and ROC curves appears similar as well, see 5.3. However, the AUC of the model with the ignore-skip rule is 0.01 higher.

Dataset	Regular model	Model with skip
Training set	0.95499	0.956451
Normal test set	0.953033	0.954555
Abnormal test set	0.147658	0.147785

Table 5.10: Average alignment scores for model with sinkcount=300 and same model with skip rule.

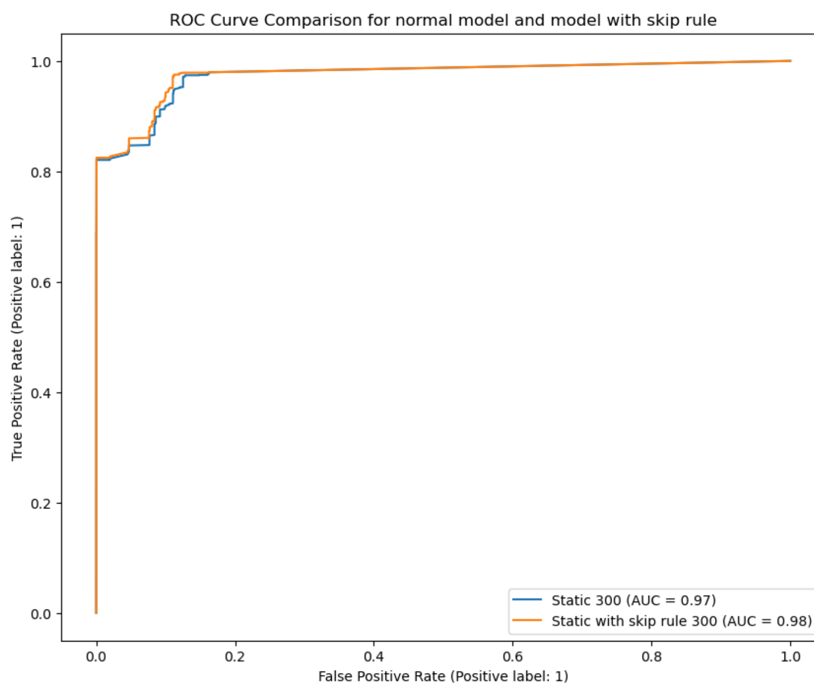
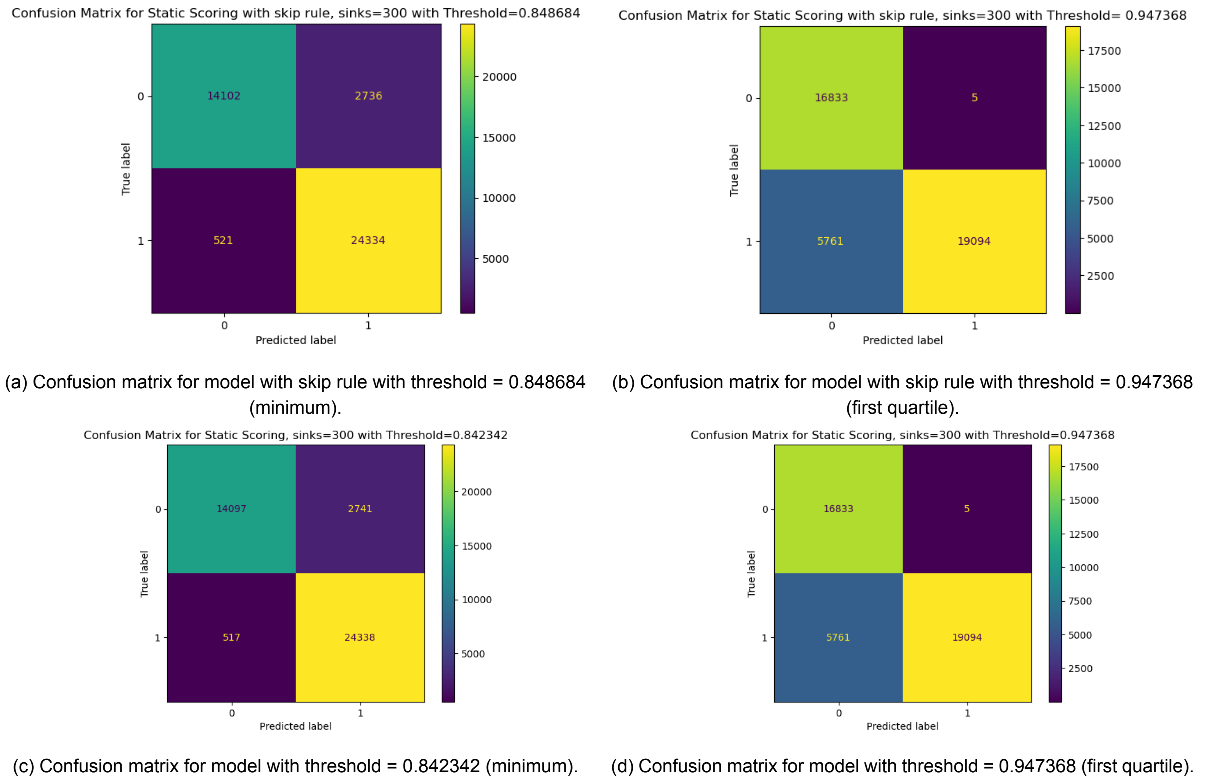


Figure 5.3: ROC curves for model with a sinkcount=300 and same model with skip rule.



(a) Confusion matrix for model with skip rule with threshold = 0.848684 (minimum).

(b) Confusion matrix for model with skip rule with threshold = 0.947368 (first quartile).

(c) Confusion matrix for model with threshold = 0.842342 (minimum).

(d) Confusion matrix for model with threshold = 0.947368 (first quartile).

Figure 5.4: Confusion matrices for two different thresholds of the modified and static models with static scoring and sinks = 300.

To further investigate, the example traces mentioned in the cases are explored:

1. 5 22 5 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21 has a alignment score of 1.0 in the normal and model with skip rule.
2. 5 5 5 22 9 11 9 11 9 26 26 11 26 23 23 23 21 21 21 has a alignment score of 0.982456 in the normal and model with skip rule.
3. 22 5 5 5 11 9 26 26 26 11 9 11 9 2 4 4 3 23 23 23 21 21 21 increases from **0.913043** in the normal model to **0.92029** in the model with skip rule
4. 5 22 5 5 11 9 11 9 11 9 26 26 26 2 23 23 23 21 21 21 has a alignment score of 0.9875 in the original normal and model with skip rule.
5. 5 5 5 22 11 9 11 9 11 9 26 26 26 3 4 3 4 3 3 23 23 23 21 21 21 has a alignment score of 0.94 in the original normal and model with skip rule.
6. 5 22 5 5 11 9 11 9 11 9 26 26 26 4 3 4 23 23 23 21 21 21 has a alignment score of 0.965909 in the original model and model with skip rule.

The results of these traces support the observation that the ignore-skip rule has a minimal effect.

5.4. Experiment 3: Modifying the original model

This experiment aims to answer research question 4: *”Can sequence alignment improve the learning process on a compact model?”*

Sequence alignment is run on the model with sinkcount=300 on the training data. Flexfringe returns a '.result' file with all aligned sequences along with their alignment scores. The traces with an alignment score of 1 are removed, leaving only traces with alignment scores less than 1. These remaining sequences are aligned using various *operations*, including:

- Jumping from one state to another
- Staying in the current state
- Skipping a state
- Skipping a symbol

These are traces from the training data, so ideally, the model should take into account the edges that are often added with sequence alignment. These operations are added as edges to the model, enhancing its accuracy and expressiveness and improving its ability to recognize normal behavior.

When adding multiple operations to a node in the model, each symbol can have at most one outgoing edge from the node, in other words, from a node there can be only one unique and unambiguous transition given a symbol. This constraint is necessary to maintain determinism in the model. Some considerations to maintain determinisim:

- One outgoing edge per symbol: Each symbol should map to a single, outgoing edge from a node. This ensures that the model behaves in a deterministic manner, where for any given symbol, there is a clear, unambiguous transition from the node.
- Existing edges with the same symbol: If the node already has an outgoing edge with a particular symbol that leads to a specific state, you cannot add another outgoing edge with the same symbol, even if it leads to a different state.
- Multiple edges with the same symbol: If multiple operations involve the same symbol, meaning multiple edges with that symbol need to be added, the edge corresponding to the most frequent transition is chosen. Only the edge representing the most common operation for that symbol is added.

To create new edges, the JSON file containing the learned model output by Flexfringe is modified by adding a new item to the *edges* field as follows:

1. *id*: set this to a string combining the source and target node IDs, formatted as: "*source node id_target node id*"
2. *source*: unique ID of the source node
3. *target*: unique ID of the target node
4. *name*: set to the symbol associated with the edge.

The source node of the newly added edge is also modified in the *nodes* field. This ensures compatibility with existing code, such as the alignment in Flexfringe. The number of times the edge has been followed is then updated in the node as follows:

1. *label* is edited, specifically "path:" in the string is modified to reflect the new occurrences of the edge.
2. *size* is updated by adding the new occurrences.

3. *path_counts* is updated by adding the new occurrences.

4. *total_paths* is updated by adding the new occurrences.

5. *trans_counts* updated by adding the new occurrences for the symbol of the edge.

5.4.1. Results

The average alignment scores for the modified model appears to be slightly higher than the and regular model with sinks=300, see Table 5.11.

Dataset	Regular model	Modified model
Training set	0.95499	0.967844
Normal test set	0.953033	0.96367
Abnormal test set	0.147658	0.159398

Table 5.11: Average alignment scores with different scoring for three AIC models, normal and modified, on the training set

Based on the ROC curves, the modified model shows improved sensitivity and specificity, as indicated by its closer proximity to the top left corner in Figure 5.5. However, the overall performance of the two models is similar, as both have identical AUC scores of 0.97.

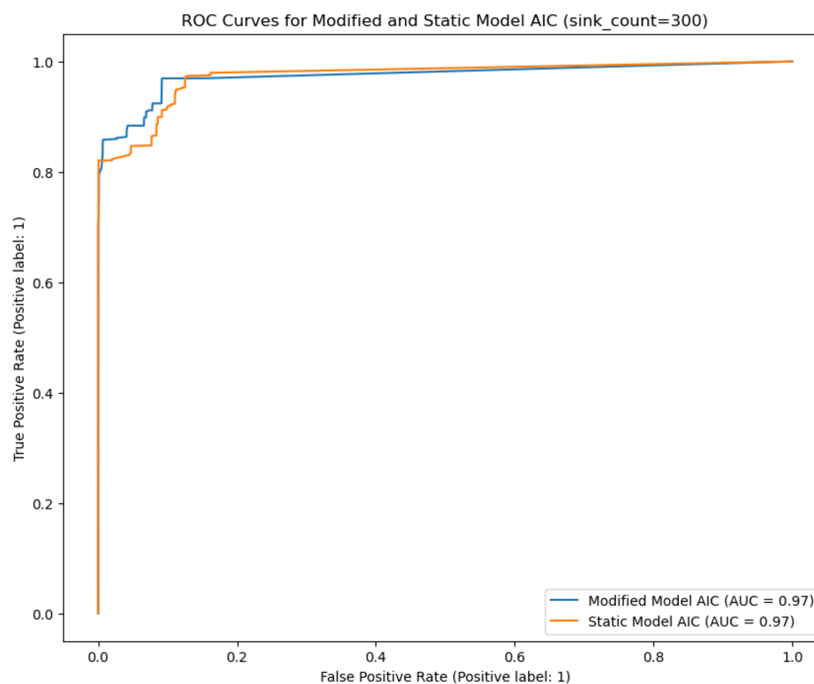
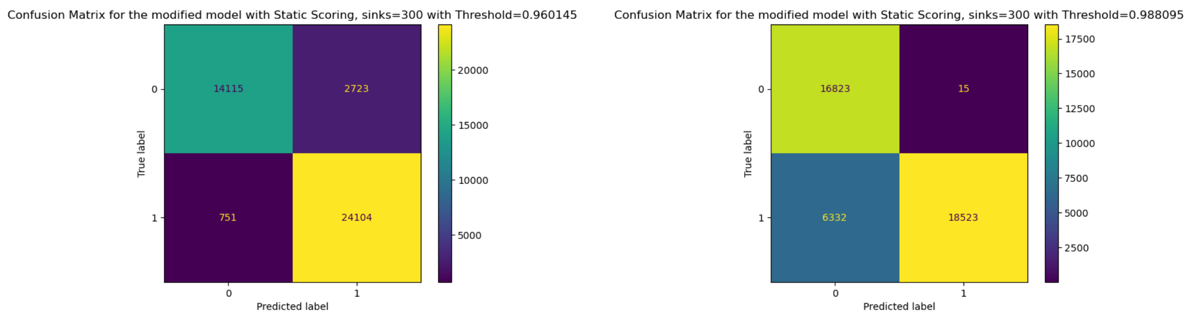


Figure 5.5: ROC curve for modified model AIC with static scoring and sinks=300.

Based on the confusion matrices, see Figure 5.6, results are similar to the results of experiment 1, see Figure 5.4c and Figure 5.4d.

Upon further inspection of the aligned sequences, it appears that modifying the model by adding the most frequently followed edges only slightly increases the alignment score of some traces. To illustrate this, we will examine some examples:

1. 5 5 5 22 11 9 11 9 26 26 11 9 26 4 3 4 2 2 23 23 23 21 21 21 from the *normal test set* has a alignment score of 0.947917 in the normal and 0.996528 in the modified mode.



(a) Confusion matrix for modified model with threshold = 0.960145 (minimum).

(b) Confusion matrix for modified model with threshold = 0.988095 (first quartile).

Figure 5.6: Confusion matrices for two different thresholds of the modified models with static scoring and sinks = 300.

2. 22 5 5 5 26 26 26 11 9 11 9 11 9 3 2 2 23 23 23 21 21 21 from the *training set* has a alignment score of 0.920455 in the normal and 0.977273 in the modified mode.
3. 22 5 5 5 26 26 11 9 11 9 11 9 26 3 3 4 3 3 4 3 3 4 3 3 4 3 23 23 23 21 21 21 from the *training set* has a alignment score of 0.88172 in the normal and 0.989247 in the modified mode.

Upon manual inspection, it appears that most traces increase their alignment score significantly, but some traces receive an alignment score of 0 in the modified model, despite achieving a good score in the normal model. For example, ”5 5 5 22 11 9 11 9 26 26 11 9 26 2” decreases from 0.982143 (normal) to 0 (modified). This is due to traces not ending in a final state as a consequence of the addition of new edges, for example:

1. 5 5 22 5 11 9 11 9 26 26 11 9 26 5 22
   state sequence modified model: [1,4,12,18,29,50,88,147,234,348,485,645,820,645/+,**645**]
   state sequence normal model: [1,4,12,18,29,50,88,147,234,348,485,645,820,820/-,820/-]
   ,where 645 is a non-final state.
2. 14 5 5 5 22 11 9 11 9 11 9 26 26 26 2
   state sequence modified model: [1,4,10,18,29,50,88,147,232,345,482,645,820,**482/+**]
   state sequence normal model: [1,4,10,18,29,50,88,147,232,345,482,645,820,820/-]
   ,where 482 is a non-final state.

6 Discussion

Results show most models with different sink counts across different scoring types achieve average alignment scores higher than 0.9 on normal data and all models achieve scores lower than 0.16 on abnormal data. ROC curves and AUC scores indicate an almost perfect classifier. Across the different scoring types, static and linear scoring methods appear to perform similar with regards to their average alignment scores on the dataset, F1-scores, ROC curves and AUC scores. Dynamic scoring obtains the best F1-scores out of all three scoring types for models with sink counts < 300, but significantly drops performance when sinkcount=300. However, given the ROC curves and AUC scores, dynamic scoring performs worse than static and linear scoring.

When comparing the implemented NW-alignment with normal predict and `predictalign` from Flexfringe. NW-alignment outperforms regular predict when ran on sinks 5, 10, 50, 100 and 300 based on ROC curves and AUC scores. However, `predictalign` still performs better given F1-scores for models with lower sink counts, specifically up until sinkcount=300. With regards to ROC curves and AUC scores, see Appendix B.4, `predictalign` and NW-alignment perform similar, until sinkcount=300. NW-alignment outperforms both predict and `predictalign` when sinkcount=300. These findings address the first research question, showing that sequence alignment with the NW-algorithm is effective in distinguishing normal and abnormal behavior in comparison to predict, but only outperforms `predictalign` when sink counts are larger.

The implementation of the ignore-skip rule to aid in the detection of parallelism does not significantly improve the behavioral differentiation that sequence alignment is able to achieve on its own given the alignment scores, F1-score and ROC curves, answering the second and third research question. While it may slightly increase alignment scores for certain traces, the improvement is minimal. The distance within which events are allowed to be ignored was arbitrarily chosen. Exhaustively searching for an optimal distance, tailored to the use case of the data, might yield better results given that the alignment scores slightly increases and the AUC score increased with 0.01.

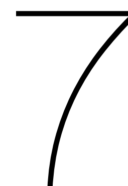
Modifying the original learned DFA with operations often taken in sequence alignment, increases alignment scores slightly. It can most likely be improved more if the issue of traces ending in a non-final state after model modification due to the addition of new edges can be solved.

A limitation of this study is that the experiments were conducted using only a single dataset. In the HDFS dataset, many traces in the abnormal set can be easily filtered out because they either end in a non-final state or contain unknown symbols. It would be ideal to evaluate how the NW-alignment performs without these traces. Additionally, the anomalous test set includes many short traces, as

noted during data exploration, and this factor was not considered. Consequently, these short traces may achieve unusually high alignment scores.

Future research could address these limitations by considering the length of traces and evaluating the performance of the NW-alignment on traces that do not contain unknown symbols or end in a non-final state. Assessing the sequence alignment algorithm on diverse datasets could demonstrate its robustness. Additionally, tuning various parameters—such as penalties, sink counts, and distances for the ignore-skip rule could potentially enhance the model's performance. The current model modifications did not include visualizing the modified graph, which could provide valuable insights into the behavior and performance of the model. Further exploration of the ignore-skip rule and model modifications is recommended.

A great addition would be to incorporate visualization of the aligned state sequences output by the algorithm. This can aid experts in identifying and differentiating behavioral patterns more effectively in practice, similar to [32] and [5].



Conclusion

The primary goal of this thesis was to investigate behavioral differencing in software logs using Finite State Automata (FSA). Anomaly detection can be approached with state machine learning by learning a model from data and then assessing whether new data fits the learned model. However, as system logs contain enormous amounts of data, among which parallel behavior, the size and complexity of the learned model can make it difficult to interpret, making it challenging for experts to analyze the model's behavior. The option to allow for sinks can enhance readability, but compromises on the model's behaviour to differentiate normal and anomalous behavior as normal data now does not conform with the model anymore.

To address these issues and allow for more nuanced behavioral differencing, where deviations in sequences are expected and allowed, the concept of sequence alignment was applied. A sequence to sequence alignment algorithm performing global alignment known as Needleman-Wunsch was adapted to perform alignment between an input sequence and state machine model. This had previously also been expored [32].

For automaton learning, the popular tool Flexfringe was used. Several Deterministic Finite Automata (DFA) with the AIC heuristic and varying sink counts were learned from the Hadoop Distributed File System (HDFS) dataset, which was processed into the Abbadingo format. The Needleman-Wunsch alignment algorithm was integrated into the Flexfringe codebas along with the parameters `--predictalignnw` and `--nwscoring` to toggle the Needleman-Wunsch (NW) alignment algorithm and scoring types.

Three scoring methods were explored: *static*, *linear*, and *dynamic*. *Static scoring* applies fixed penalties for matches, mismatches, and gaps. *Linear scoring* adjusts the penalty according to the size of the gap in the aligned sequence, and *dynamic scoring* takes into account the distance between nodes. To improve the detection of parallel behaviors, an ignore-skip rule was introduced, which allows events to reoccur at no cost if they had been previously skipped within a distance of 3 events ago. Lastly, model modifications were made by incorporating frequently performed operations during sequence alignment.

Results show sequence alignment effectively detects flows that deviate from the expected behavior given average alignment scores per dataset, normal traces achieved an average alignment score above 0.9 across all models with static and linear scoring and abnormal traces received an average alignment score below 0.16 across all models and scoring types. Among the scoring methods, static scoring and linear scoring performed similar given their F1-scores (>0.93). Dynamic scoring obtained high F1-score

(>0.95) for sinkcount=50 and sinkcount=100, but performance dropped at sink count=300.

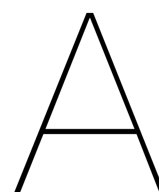
Adding the ignore-skip rule was not able to recognize parallel processes and improve performance. However, NW-alignment is able to outperform regular prediction with models with sink counts larger than 5. Nevertheless, `predictalign` from Flexfringe outperforms NW-alignment and predict given the F1-score, for models with sink counts up to 100. Based on results, we can conclude that NW-alignment can improve results when used on models with larger sink counts.

Lastly, modifying the model with operations commonly used in NW-alignment did not enhance the learning process. However, these modifications, along with the ignore-skip rule, can be refined and further explored in future work.

The code used in this research is available for review and replication. It can be accessed on GitHub [1]. The repository includes the NW-alignment algorithm, datasets, results, and documentation needed to reproduce the results presented in this thesis.

In summary, this thesis shows that incorporating sequence alignment techniques into FSA-based models can enhance the detection of behavioral differences in software logs, especially when models are learnt with larger sinks. The findings suggest that further refinement of penalty selection, scoring methods and alignment operations could yield even more insights for software behavior analysis.

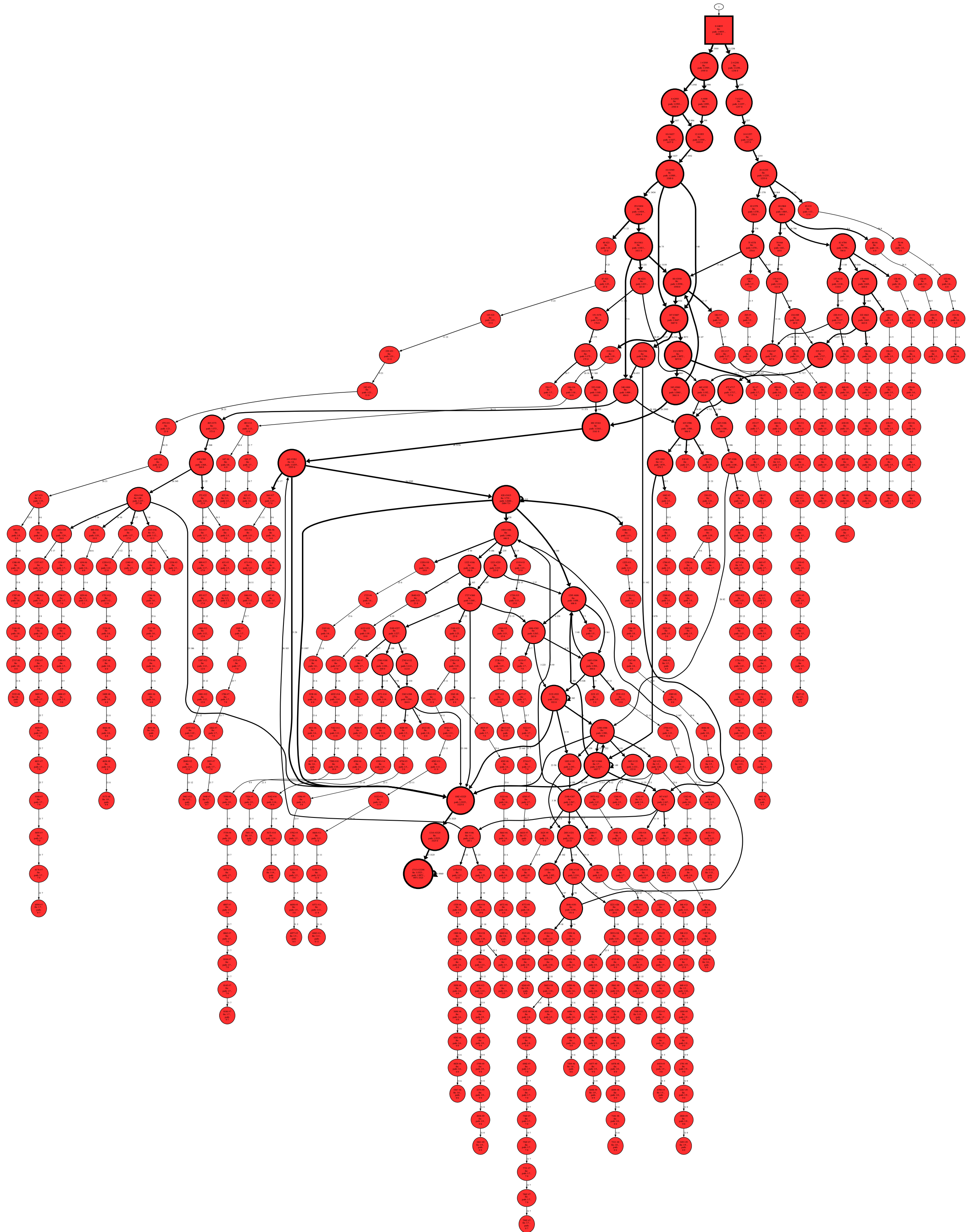
[1] https://github.com/Mirijam1/NW-sequence-alignment



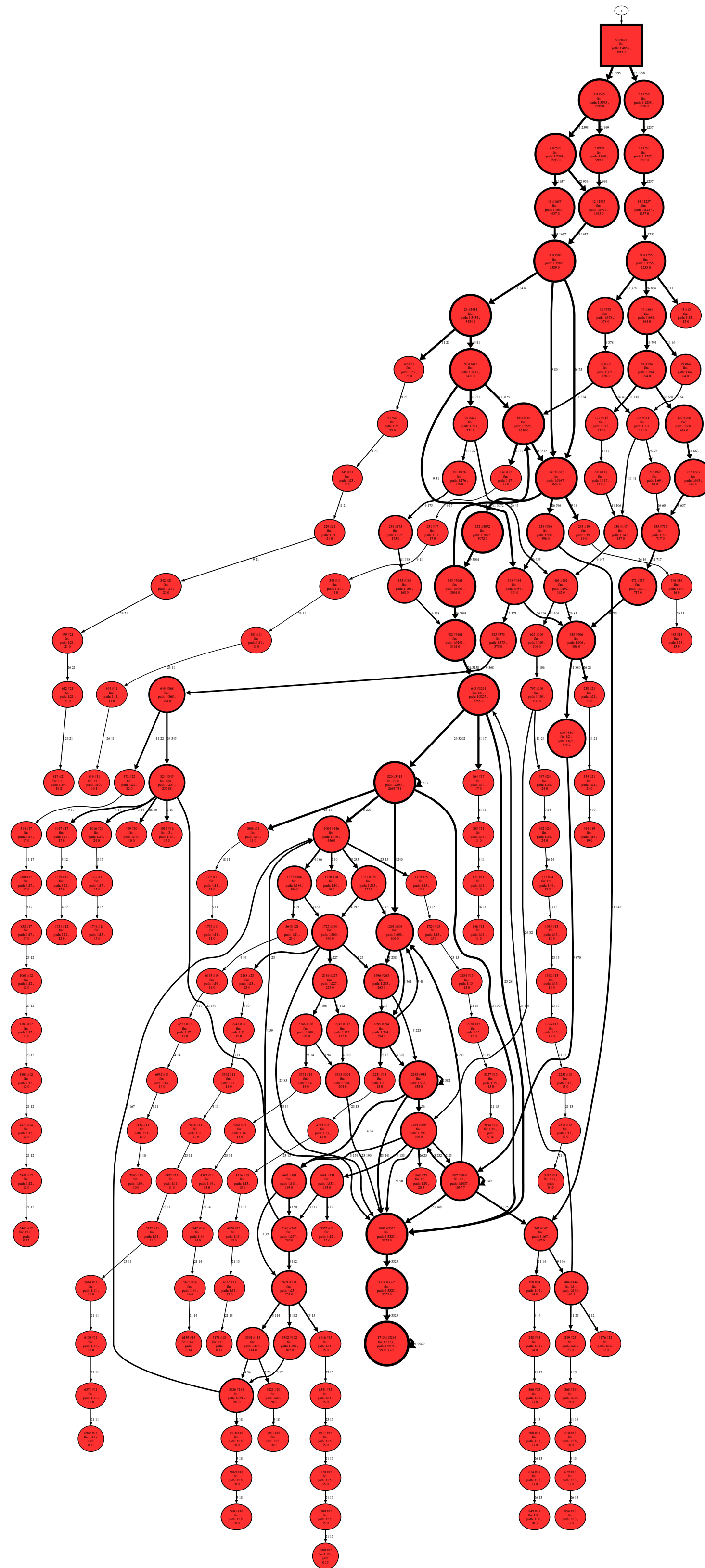
Appendix A

A.1. Learned FSA

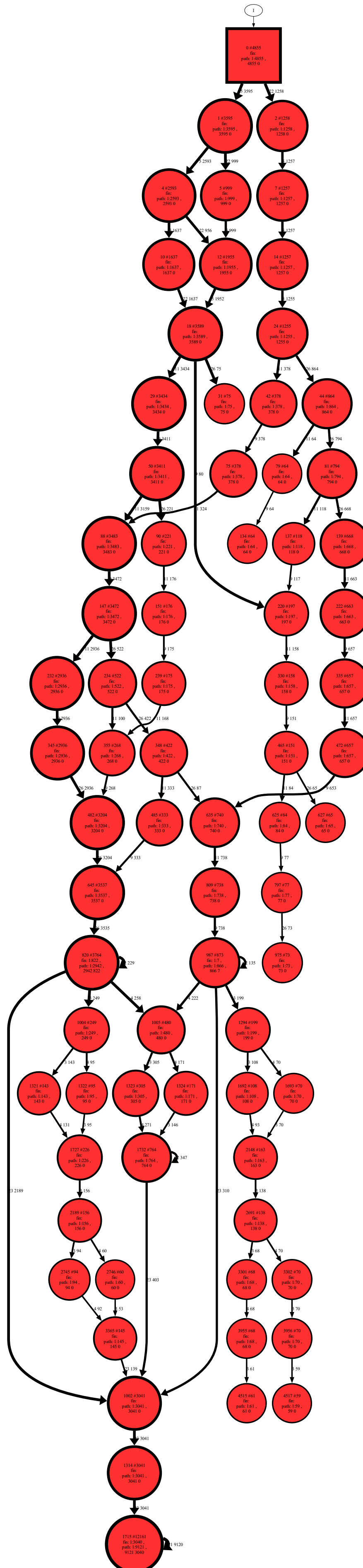
A.1.1. HDFS with AIC and sink count=5



A.1.2. HDFS with AIC and sink count=10

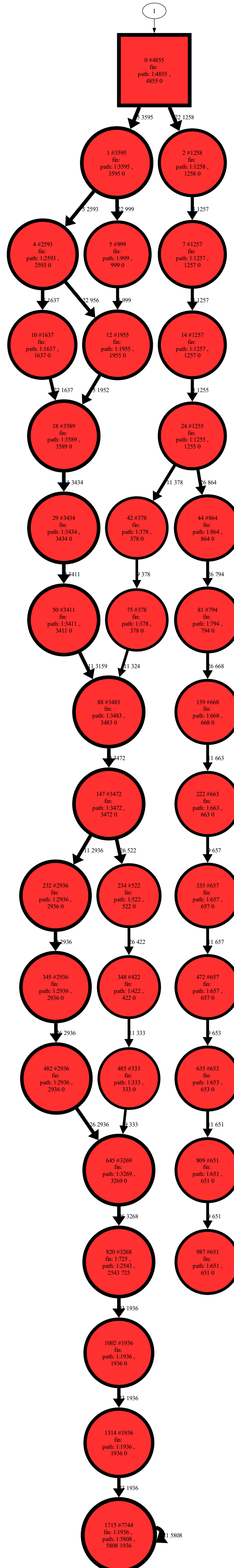


A.1.3. HDFS with AIC and sink count=50

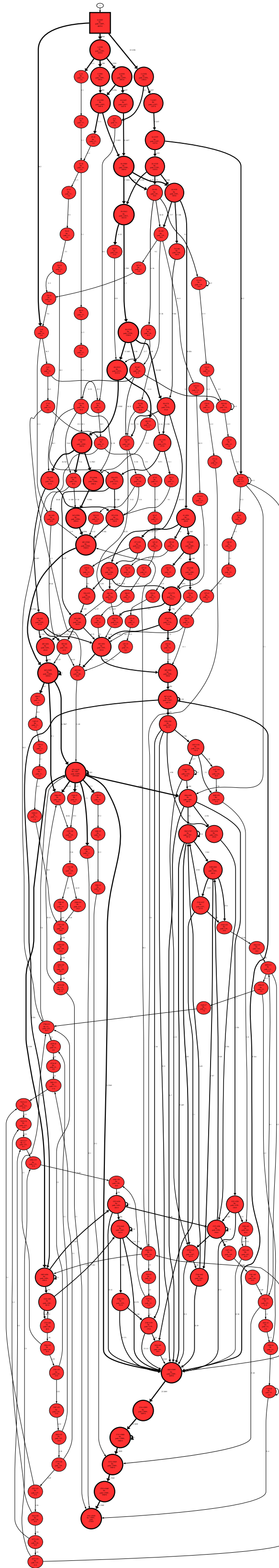


A.1.4. HDFS with AIC and sink count=100

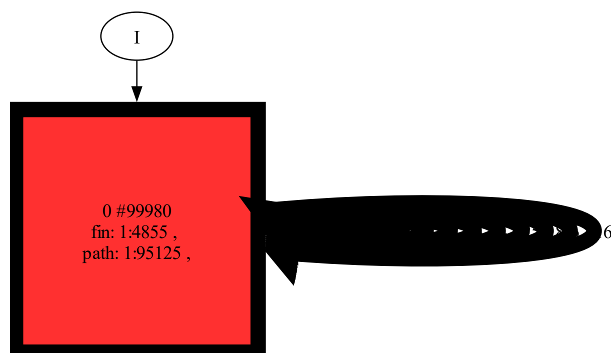
A.1.5. HDFS with AIC and sink count=300



A.1.6. HDFS with AIC.ini



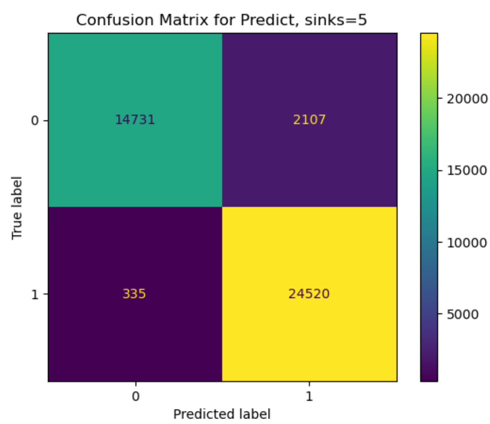
A.1.7. HDFS with edsm.ini



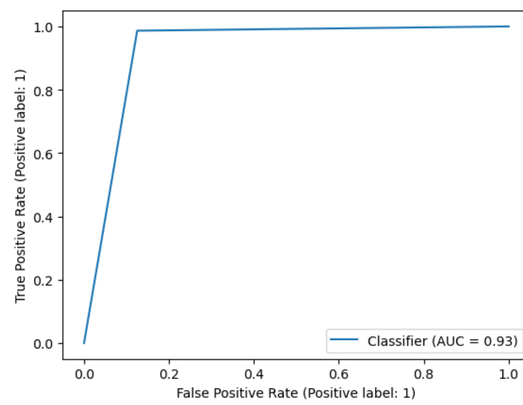
B

Appendix B

B.1. Confusion Matrices for predict

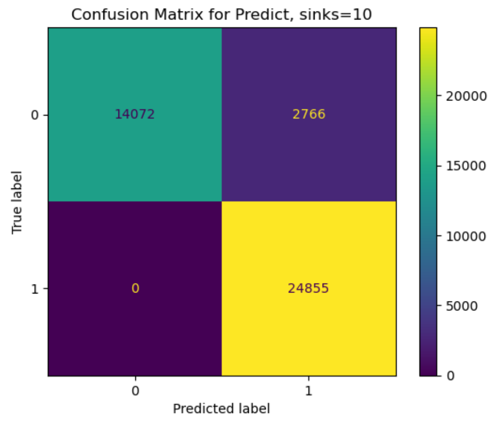


(a) Confusion matrix for predict with model with sinks=5.

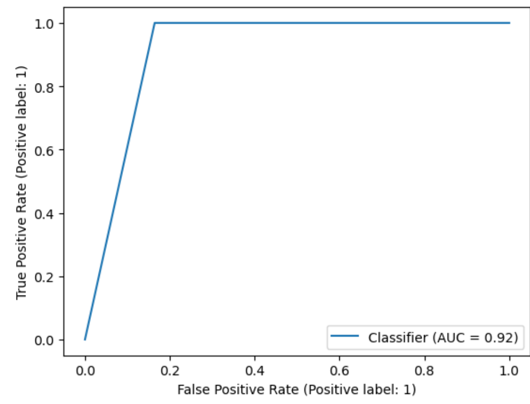


(b) ROC curve for predict with model with sink=5.

Figure B.1: Confusion matrix and ROC curve for predict with model with sinks=5

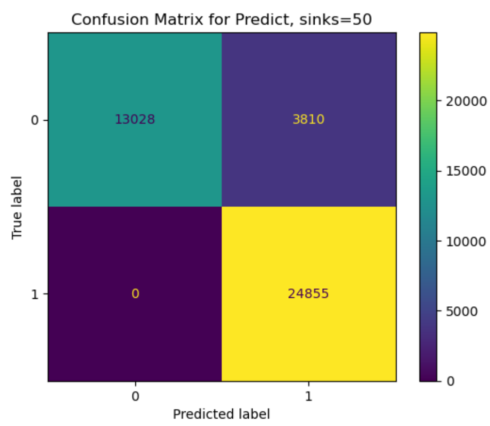


(a) Confusion matrix for predict with model with sinks=10.

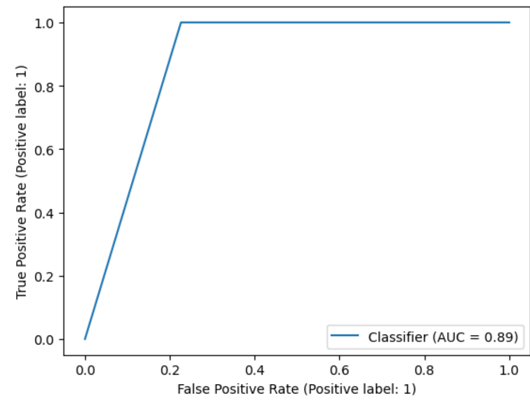


(b) ROC curve for predict with model with sink=10.

Figure B.2: Confusion matrix and ROC curve for predict with model with sinks=10

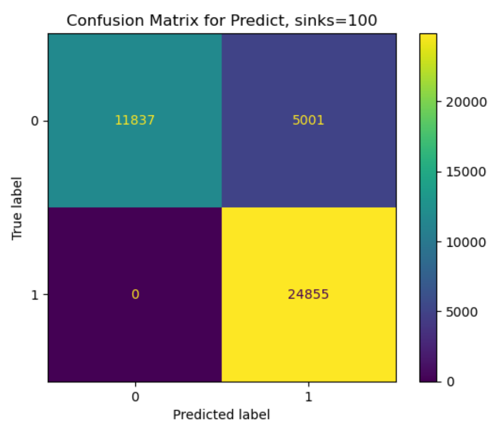


(a) Confusion matrix for predict with model with sinks=50.

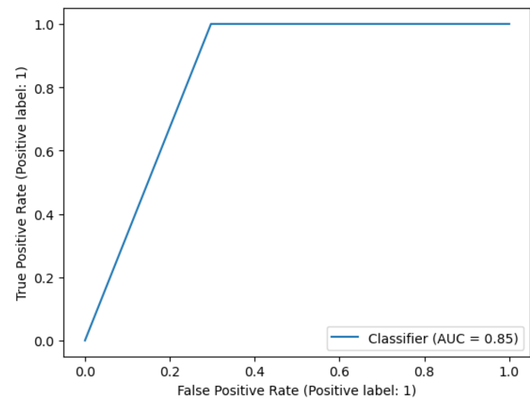


(b) ROC curve for predict with model with sink=50.

Figure B.3: Confusion matrix and ROC curve for predict with model with sinks=50

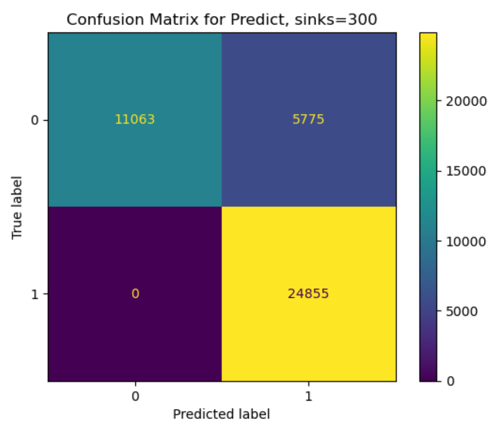


(a) Confusion matrix for predict with model with sinks=100.

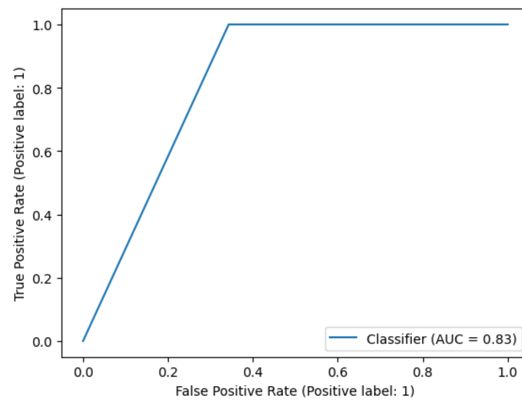


(b) ROC curve for predict with model with sink=100.

Figure B.4: Confusion matrix and ROC curve for predict with model with sinks=100



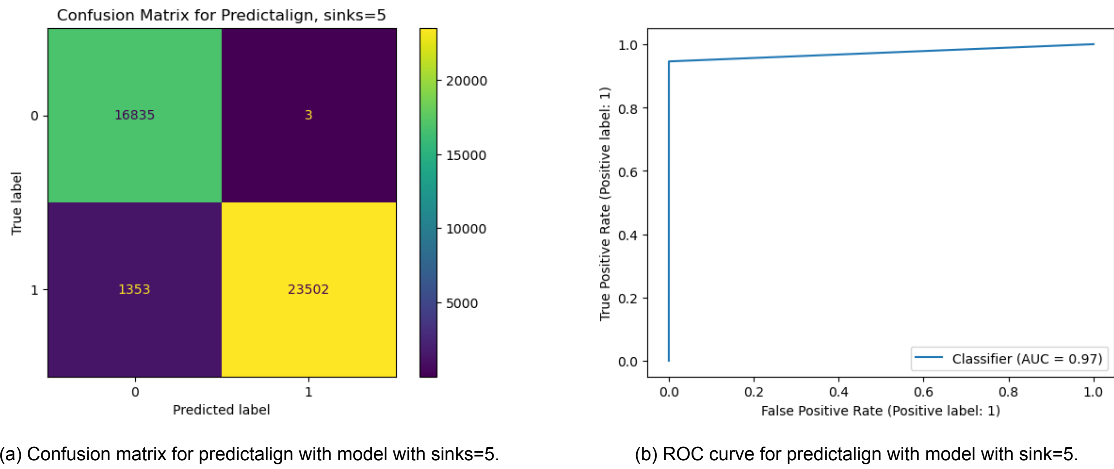
(a) Confusion matrix for predict with model with sinks=300.



(b) ROC curve for predict with model with sink=300.

Figure B.5: Confusion matrix and ROC curve for predict with model with sinks=300

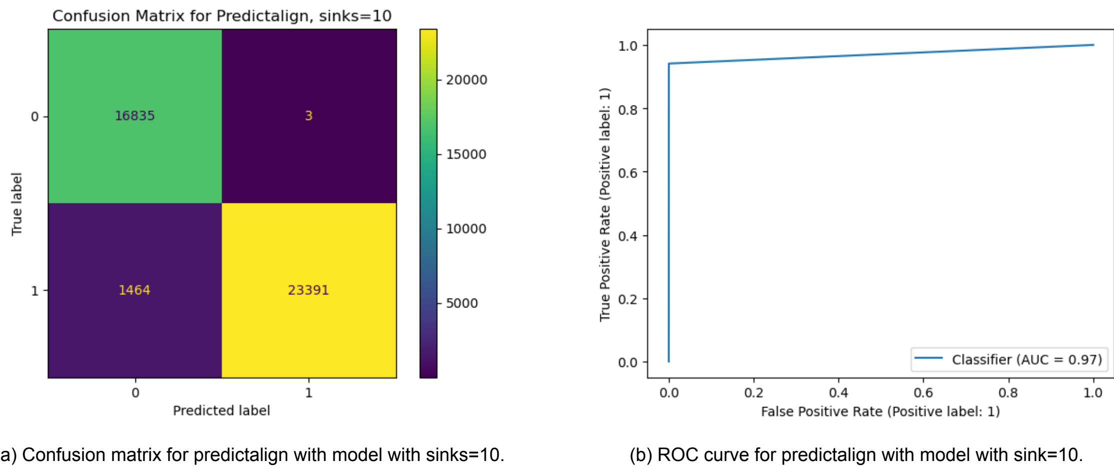
B.2. Confusion Matrices for predictalign



(a) Confusion matrix for predictalign with model with sinks=5.

(b) ROC curve for predictalign with model with sink=5.

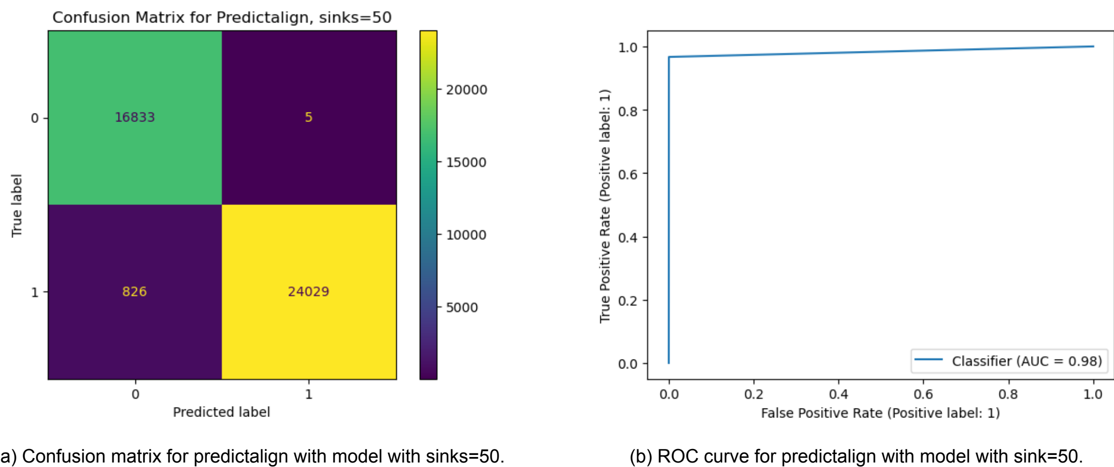
Figure B.6: Confusion matrix and ROC curve for predictalign with model with sinks=5



(a) Confusion matrix for predictalign with model with sinks=10.

(b) ROC curve for predictalign with model with sink=10.

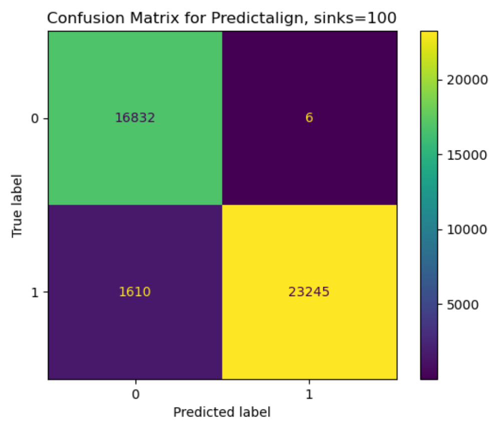
Figure B.7: Confusion matrix and ROC curve for predictalign with model with sinks=10



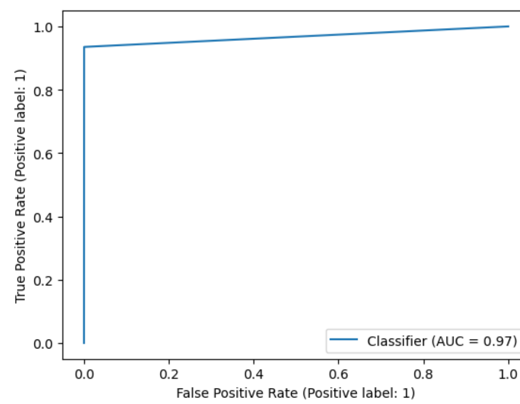
(a) Confusion matrix for predictalign with model with sinks=50.

(b) ROC curve for predictalign with model with sink=50.

Figure B.8: Confusion matrix and ROC curve for predictalign with model with sinks=50

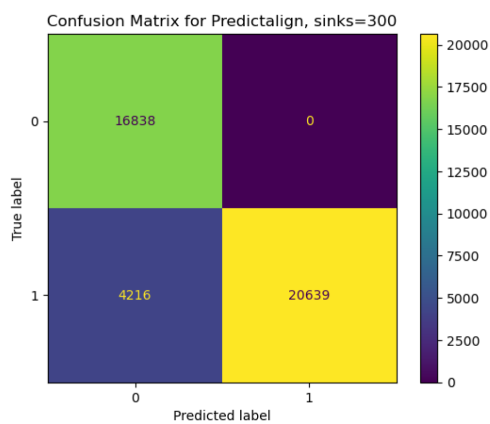


(a) Confusion matrix for predictalign with model with sinks=100.

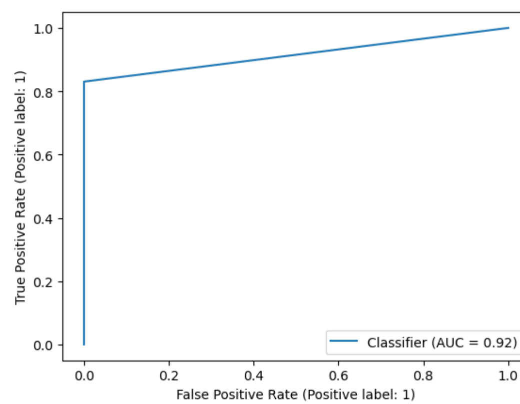


(b) ROC curve for predictalign with model with sink=100.

Figure B.9: Confusion matrix and ROC curve for predictalign with model with sinks=100



(a) Confusion matrix for predictalign with model with sinks=300.

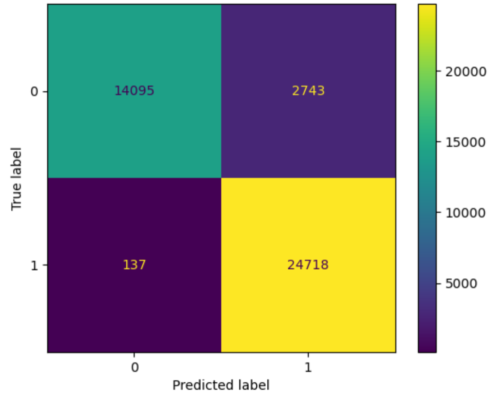


(b) ROC curve for predictalign with model with sink=300.

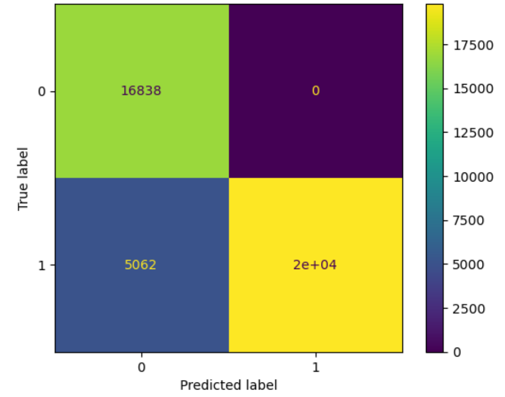
Figure B.10: Confusion matrix and ROC curve for predictalign with model with sinks=300

B.3. Confusion matrices for Needleman-Wunsch alignment

B.3.1. AIC model, sinks=50

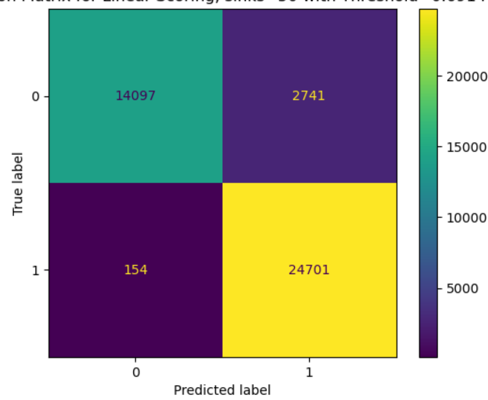


(a) Confusion matrix for model with threshold=0.842342 (minimum).

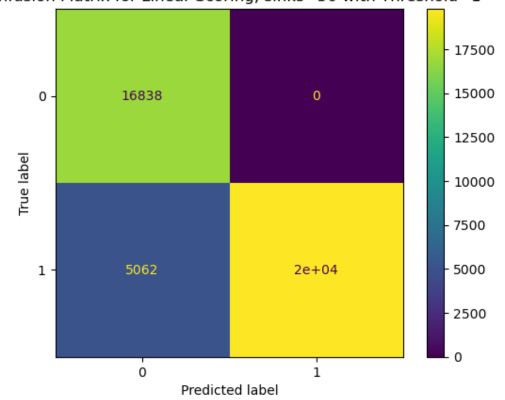


(b) Confusion matrix for model with threshold=1.0 (first quartile).

Figure B.11: Confusion matrices for model with sinks=50 and static scoring with two thresholds

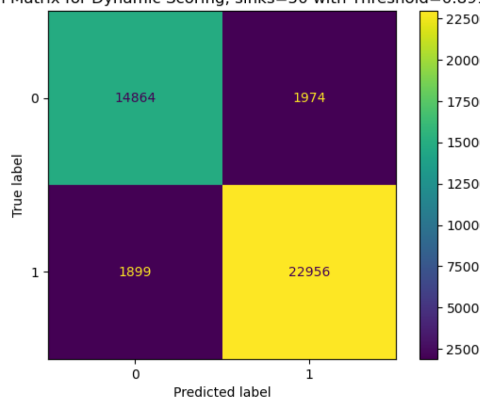


(a) Confusion matrix for model with threshold=0.691441 (minimum).

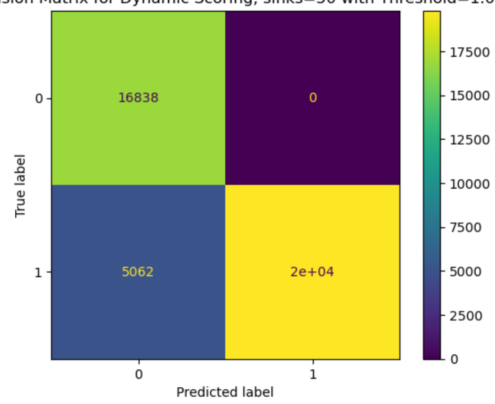


(b) Confusion matrix for model with threshold=1.0 (first quartile).

Figure B.12: Confusion matrices for model with sinks=50 and linear scoring with two thresholds



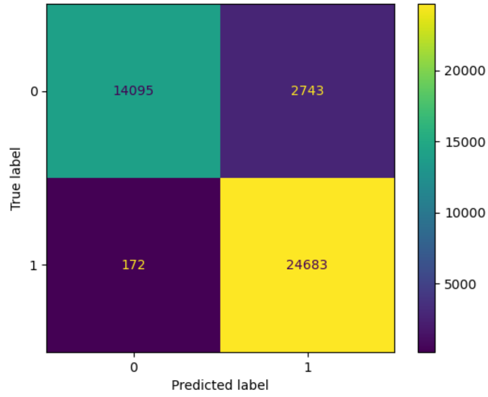
(a) Confusion matrix for model with threshold=0.891892 (minimum).



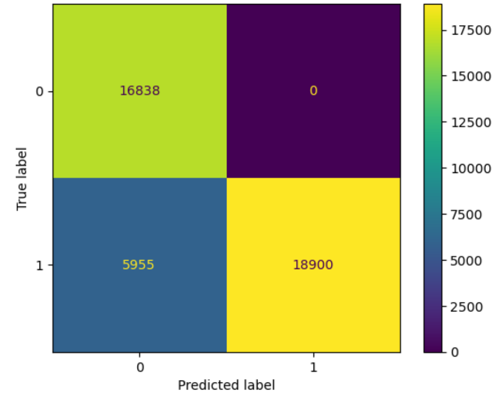
(b) Confusion matrix for model with threshold=1.0 (first quartile).

Figure B.13: Confusion matrices for model with sinks=50 and dynamic scoring with two thresholds

B.3.2. AIC model, sinks=100

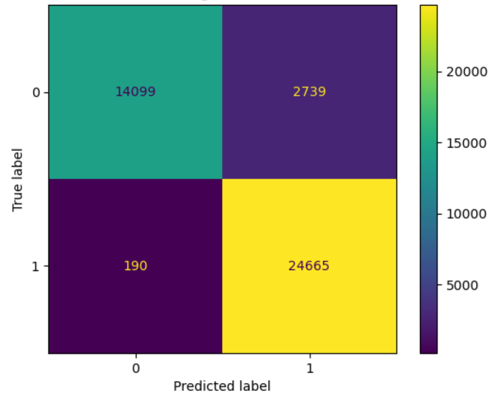


(a) Confusion matrix for model with threshold=0.842342 (minimum).

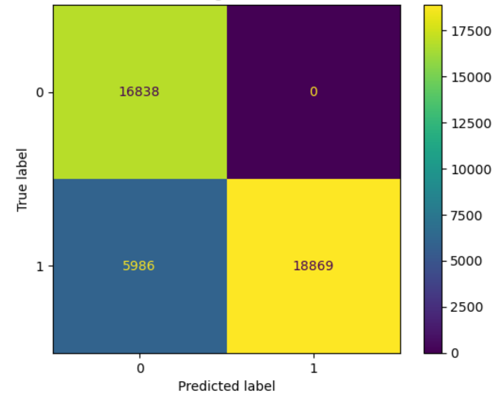


(b) Confusion matrix for model with threshold=0.9875 (first quartile).

Figure B.14: Confusion matrices for model with sinks=100 and static scoring with two thresholds

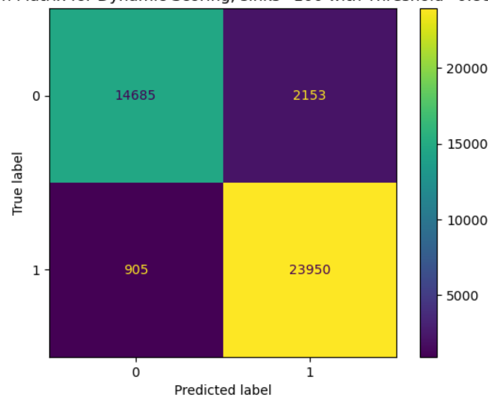


(a) Confusion matrix for model with threshold=0.691441 (minimum).

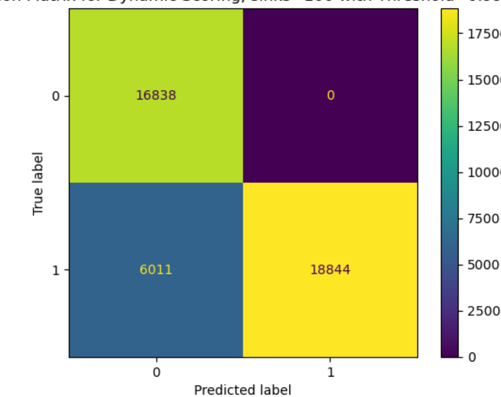


(b) Confusion matrix for model with threshold=0.9875 (first quartile).

Figure B.15: Confusion matrices for model with sinks=100 and linear scoring with two thresholds



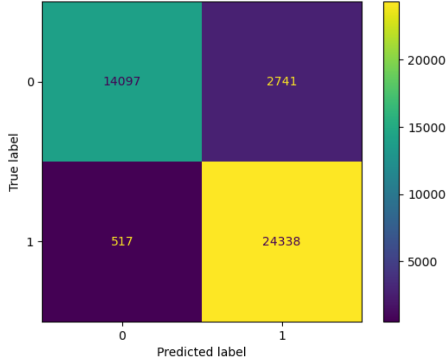
(a) Confusion matrix for model with threshold=0.8875 (minimum).



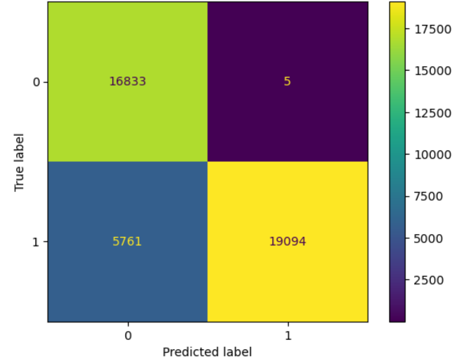
(b) Confusion matrix for model with threshold=0.991667 (first quartile).

Figure B.16: Confusion matrices for model with sinks=100 and dynamic scoring with two thresholds

B.3.3. AIC model, sinks=300

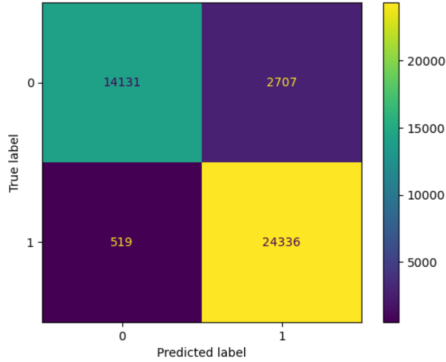


(a) Confusion matrix for model with threshold=0.842342 (minimum).

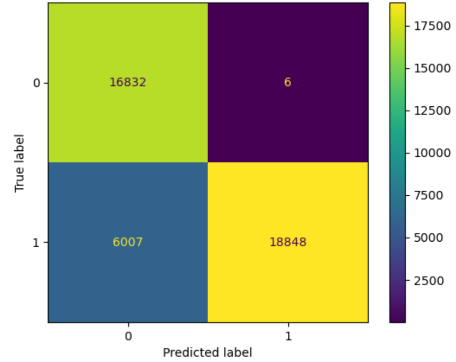


(b) Confusion matrix for model with threshold=0.947368 (first quartile).

Figure B.17: Confusion matrices for model with sinks=300 and static scoring with two thresholds

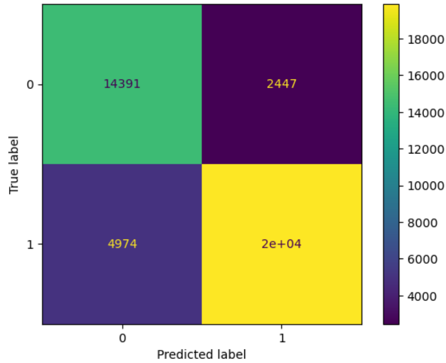


(a) Confusion matrix for model with threshold=0.361111 (minimum).

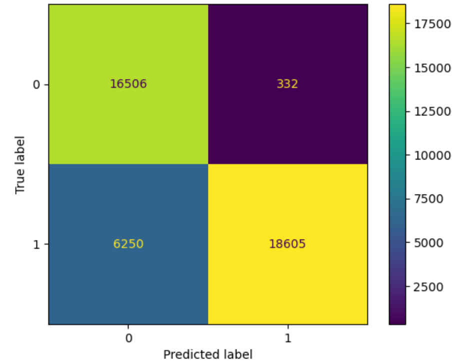


(b) Confusion matrix for model with threshold=0.929825 (first quartile).

Figure B.18: Confusion matrices for model with sinks=300 and linear scoring with two thresholds



(a) Confusion matrix for model with threshold=0.878378 (first quartile).



(b) Confusion matrix for model with threshold=0.95614 (first quartile).

Figure B.19: Confusion matrices for model with sinks=300 and dynamic scoring with two thresholds

B.4. ROC curves

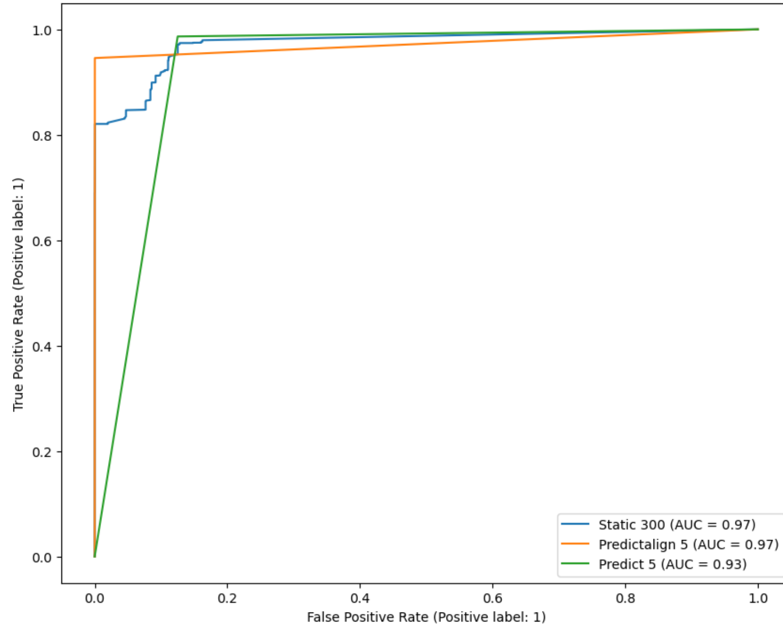


Figure B.20: ROC curves for model with sinks=5 with predict, predictalign and a model with sinks=300 with NW-alignment and static scoring

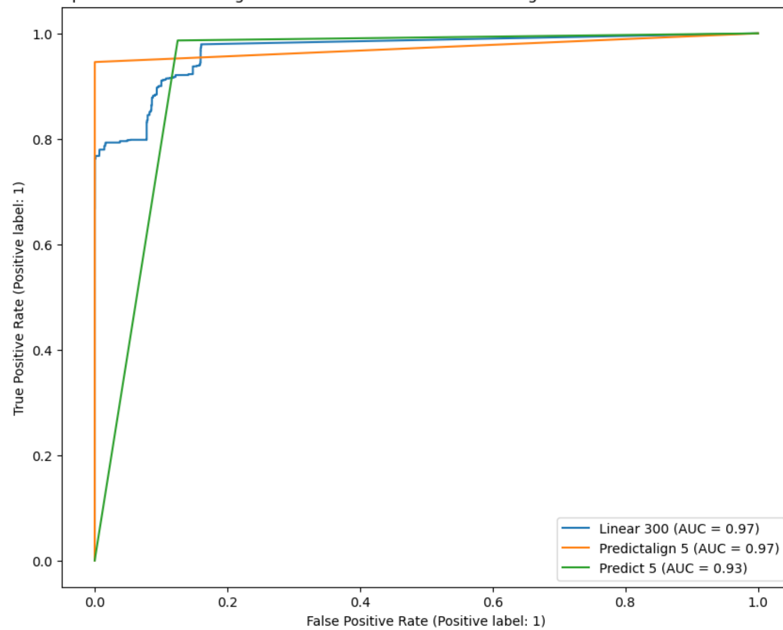


Figure B.21: ROC curves for model with sinks=5 with predict, predictalign and a model with sinks=300 with NW-alignment and linear scoring

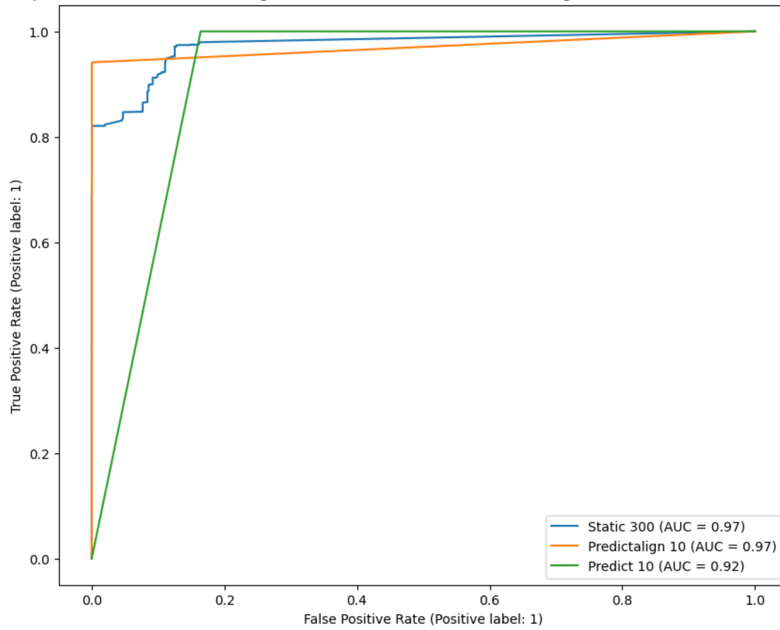


Figure B.22: ROC curves for model with a sinks=10 with predict, predictalign and a model with a sinks=300 with NW-alignment and static scoring

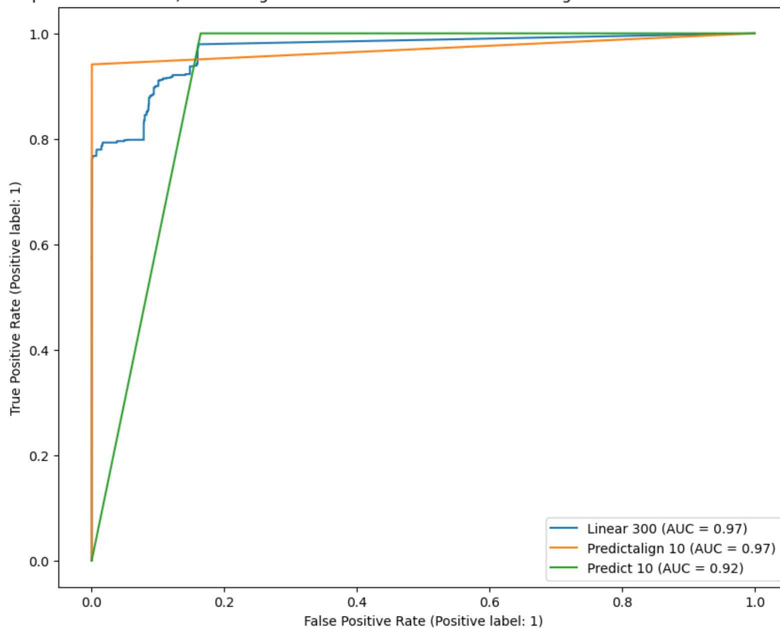


Figure B.23: ROC curves for model with a sinks=10 with predict, predictalign and a model with a sinks=300 with NW-alignment and linear scoring

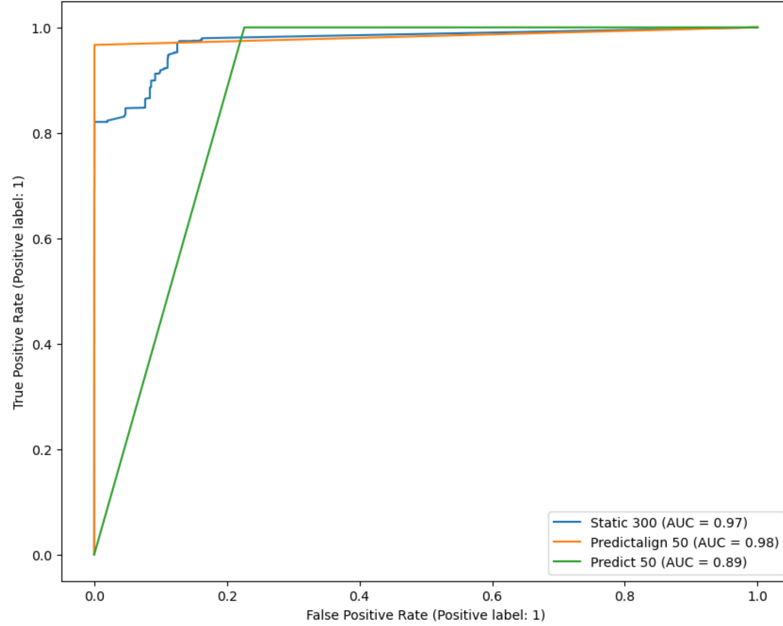


Figure B.24: ROC curves for model with a sinks=50 with predict, predictalign and a model with a sinks=300 with NW-alignment and static scoring

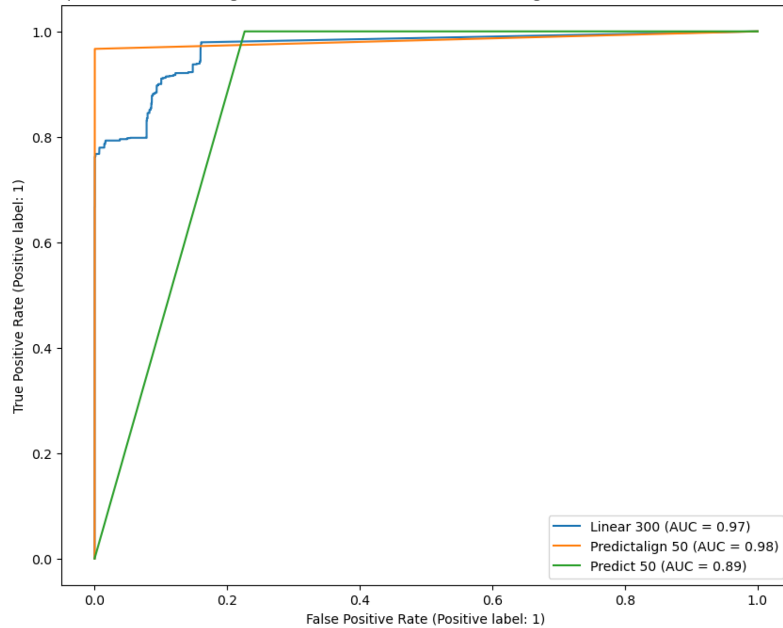


Figure B.25: ROC curves for model with a sinks=50 with predict, predictalign and a model with a sinks=300 with NW-alignment and linear scoring

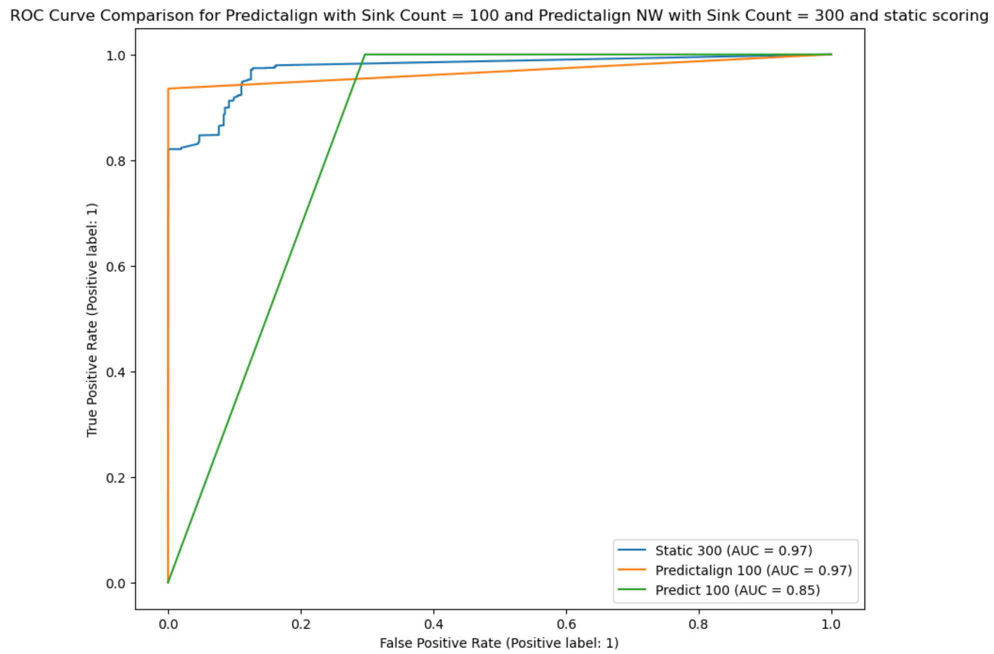


Figure B.26: ROC curves for model with a sinks=100 with predict, predictalign and a model with a sinks=300 with NW-alignment and static scoring

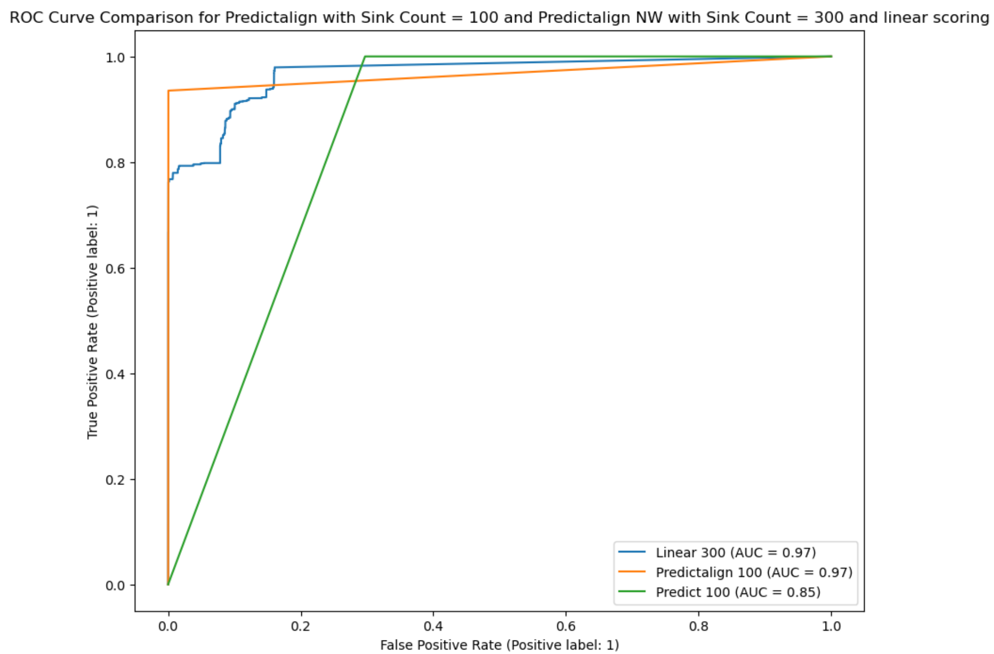


Figure B.27: ROC curves for model with a sinks=100 with predict, predictalign and a model with a sinks=300 with NW-alignment and linear scoring

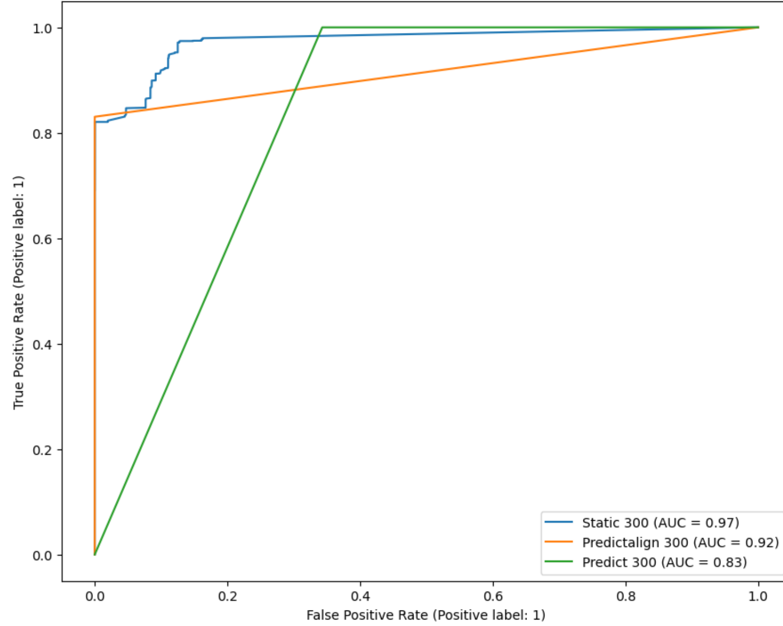


Figure B.28: ROC curves for model with a sinks=300 with predict, predictalign and a model with a sinks=300 with NW-alignment and static scoring

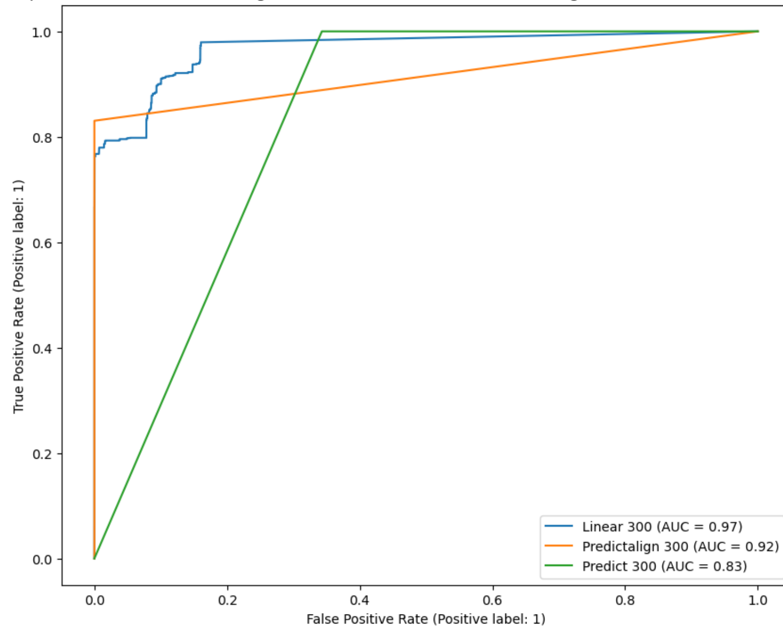


Figure B.29: ROC curves for model with a sinks=300 with predict, predictalign and a model with a sinks=300 with NW-alignment and linear scoring

Bibliography

[1] W. U. Hassan, S. Guo, D. Li, *et al.*, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage," en, in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019, isbn: 978-1-891562-55-6. doi: 10.14722/ndss.2019.23349. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_03B-1-3_UlHassan_paper.pdf (visited on 01/18/2024).

[2] T. V. Ede, H. Aghakhani, N. Spahn, *et al.*, "DEEPCASE: Semi-Supervised Contextual Analysis of Security Events," en, in *2022 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2022, pp. 522–539, isbn: 978-1-66541-316-9. doi: 10.1109/SP46214.2022.9833671. [Online]. Available: https://ieeexplore.ieee.org/document/9833671/ (visited on 12/03/2023).

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," en, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky Montana USA: ACM, Oct. 2009, pp. 117–132, isbn: 978-1-60558-752-3. doi: 10.1145/1629575.1629587. [Online]. Available: https://dl.acm.org/doi/10.1145/1629575.1629587 (visited on 09/28/2023).

[4] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, 15:1–15:58, Jul. 2009, issn: 0360-0300. doi: 10.1145/1541880.1541882. [Online]. Available: https://doi.org/10.1145/1541880.1541882 (visited on 06/05/2023).

[5] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz, "Using finite-state models for log differencing," en, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 49–59, isbn: 978-1-4503-5573-5. doi: 10.1145/3236024.3236069. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3236069 (visited on 05/29/2023).

[6] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," en, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA: ACM, Oct. 2017, pp. 1285–1298, isbn: 978-1-4503-4946-8. doi: 10.1145/3133956.3134015. [Online]. Available: https://dl.acm.org/doi/10.1145/3133956.3134015 (visited on 12/19/2023).

[7] E. Lomagin, "Anomaly Detection in System Event Logs," en, Ph.D. dissertation, 2019.

[8] K. Xu, Z.-L. Zhang, and S. Bhattacharyya, "Profiling internet backbone traffic: Behavior models and applications," in *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '05, New York, NY, USA: Association for Computing Machinery, Aug. 2005, pp. 169–180, isbn: 978-1-59593-009-5. doi: 10.1145/1080091.1080112. [Online]. Available: https://dl.acm.org/doi/10.1145/1080091.1080112 (visited on 06/19/2024).

[9] W. Aalst, *Process Mining: Data Science in Action*. Jan. 2016, isbn: 978-3-662-49850-7. doi: 10.1007/978-3-662-49851-4.

[10] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989, Conference Name: Proceedings of the IEEE, issn: 1558-2256. doi: 10.1109/5.24143. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/24143 (visited on 04/22/2024).

[11] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, Sep. 1977, issn: 0360-0300. doi: 10.1145/356698.356702. [Online]. Available: https://dl.acm.org/doi/10.1145/356698.356702 (visited on 04/22/2024).

[12] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, Aug. 1996, Conference Name: Proceedings of the IEEE, issn: 1558-2256. doi: 10.1109/5.533956. [Online]. Available: https://ieeexplore.ieee.org/document/533956 (visited on 11/27/2023).

[13] Graham, Dorothy, Van Veenendaal, Erik, and Evans, Isabel, *Foundations of Software Testing: ISTQB Certification*, en. Cengage Learning Business Press, 2006. [Online]. Available: https://www.abebooks.com/9781844803552/Foundations-Software-Testing-ISTQB-Certification-1844803554/plp (visited on 04/22/2024).

[14] M. J. H. Heule and S. Verwer, "Exact DFA Identification Using SAT Solvers," en, in *Grammatical Inference: Theoretical Results and Applications*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 6339, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 66–79, isbn: 978-3-642-15487-4 978-3-642-15488-1. doi: 10.1007/978-3-642-15488-1_7. [Online]. Available: http://link.springer.com/10.1007/978-3-642-15488-1_7 (visited on 03/15/2024).

[15] Vodencarevic, Asmir, Alexander Maier, and Oliver, Niggemann, "Evaluating Learning Algorithms for Stochastic Finite Automata - Comparative Empirical Analyses on Learning Models for Technical Systems:" en, in *Proceedings of the 2nd International Conference on Pattern Recognition Applications and Methods*, Barcelona, Spain: SciTePress - Science, 2013, pp. 229–238, isbn: 978-989-8565-41-9. doi: 10.5220/0004255702290238. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0004255702290238 (visited on 03/16/2024).

[16] M. J. H. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," en, *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, 2012, issn: 1573-7616. doi: 10.1007/s10664-012-9222-z. [Online]. Available: https://doi.org/10.1007/s10664-012-9222-z (visited on 04/16/2024).

[17] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," en, in *Grammatical Inference and Applications*, R. C. Carrasco and J. Oncina, Eds., Berlin, Heidelberg: Springer, 1994, pp. 139–152, isbn: 978-3-540-48985-6. doi: 10.1007/3-540-58473-0_144.

[18] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm," en, in *Grammatical Inference*, V. Honavar and G. Slutzki, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1998, pp. 1–12, isbn: 978-3-540-68707-8. doi: 10.1007/BFb0054059.

[19] W. Hoeffding, "Probability Inequalities for Sums of Bounded Random Variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963, Publisher: [American Statistical Association, Taylor & Francis, Ltd.], issn: 0162-1459. doi: 10.2307/2282952. [Online]. Available: https://www.jstor.org/stable/2282952 (visited on 03/17/2024).

[20] S. Verwer and C. Hammerschmidt, *FlexFringe: Modeling Software Behavior by Learning Probabilistic Automata*, arXiv:2203.16331 [cs] version: 1, Mar. 2022. doi: 10.48550/arXiv.2203.16331. [Online]. Available: http://arxiv.org/abs/2203.16331 (visited on 03/25/2024).

[21] S. Verwer, M. De Weerdt, and C. Witteveen, "A Likelihood-Ratio Test for Identifying Probabilistic Deterministic Real-Time Automata from Positive Data," en, in *Grammatical Inference: Theoretical Results and Applications*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 6339, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 203–216, isbn: 978-3-642-15487-4 978-3-642-15488-1. doi: 10.1007/978-3-642-15488-1_17. [Online]. Available: http://link.springer.com/10.1007/978-3-642-15488-1_17 (visited on 04/16/2024).

[22] F. Thollard, P. Dupont, and C. de la Higuera, "Probabilistic DFA Inference using Kullback-Leibler Divergence and Minimality," en, 2000. [Online]. Available: https://dial.uclouvain.be/pr/boreal/object/boreal:109443 (visited on 04/16/2024).

[23] S. Verwer, "Efficient Identification of Timed Automata: Theory and Practice," Dissertation (TU Delft), Delft, 2010. [Online]. Available: http://repository.tudelft.nl/assets/uuid:61d9f199-7b01-45be-a6ed-04498113a212/thesis_final.pdf (visited on 07/07/2024).

[24] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont, "STAMINA: A competition to encourage the development and assessment of software model inference techniques," en, *Empirical Software Engineering*, vol. 18, no. 4, pp. 791–824, Aug. 2013, issn: 1382-3256, 1573-7616. doi: 10.1007/s10664-012-9210-3. [Online]. Available: http://link.springer.com/10.1007/s10664-012-9210-3 (visited on 07/07/2024).

[25] J. A. Bergstra, A. Ponse, and S. A. Smolka, *Handbook of Process Algebra*, en. Elsevier, Mar. 2001, Google-Books-ID: gSH9zg5s3ygC, isbn: 978-0-08-053367-4.

[26] N. Walkinshaw and K. Bogdanov, "Automated Comparison of State-Based Software Models in Terms of Their Language and Structure," en, *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 2, pp. 1–37, Mar. 2013, issn: 1049-331X, 1557-7392. doi: 10.1145/2430545.2430549. [Online]. Available: https://dl.acm.org/doi/10.1145/2430545.2430549 (visited on 03/27/2024).

[27] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, and S. Vanak, "Testing methods for X-machines: A review," en, *Formal Aspects of Computing*, vol. 18, no. 1, pp. 3–30, Mar. 2006, issn: 1433-299X. doi: 10.1007/s00165-005-0085-6. [Online]. Available: https://doi.org/10.1007/s00165-005-0085-6 (visited on 04/01/2024).

[28] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, Mar. 2001, issn: 0360-0300. doi: 10.1145/375360.375365. [Online]. Available: https://doi.org/10.1145/375360.375365 (visited on 10/02/2023).

[29] V. Likic, *The Needleman-Wunsch algorithm for sequence alignment*, Melbourne. [Online]. Available: https://www.cs.sjsu.edu/~aid/cs152/NeedlemanWunsch.pdf.

[30] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," eng, *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970, issn: 0022-2836. doi: 10.1016/0022-2836(70)90057-4.

[31] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," eng, *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981, issn: 0022-2836. doi: 10.1016/0022-2836(81)90087-5.

[32] S. Tsoni, “Log Differencing using State Machines for Anomaly Detection,” en, Ph.D. dissertation, 2019. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Ab0b39832-c921-412c-b6f8-9ac4c52b57f6 (visited on 05/29/2023).

[33] *What is A Confusion Matrix in Machine Learning? The Model Evaluation Tool Explained*, en. [Online]. Available: https://www.datacamp.com/tutorial/what-is-a-confusion-matrix-in-machine-learning (visited on 07/04/2024).

[34] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, “Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis,” in *2009 Ninth IEEE International Conference on Data Mining*, ISSN: 2374-8486, Dec. 2009, pp. 149–158. doi: 10.1109/ICDM.2009.60.

[35] M. Goldstein, D. Raz, and I. Segall, “Experience Report: Log-Based Behavioral Differencing,” in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, ISSN: 2332-6549, Oct. 2017, pp. 282–293. doi: 10.1109/ISSRE.2017.14.

[36] A. W. Biermann and J. A. Feldman, “On the Synthesis of Finite-State Machines from Samples of Their Behavior,” *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 592–597, Jun. 1972, Conference Name: IEEE Transactions on Computers, issn: 1557-9956. doi: 10.1109/TC.1972.5009015. [Online]. Available: https://ieeexplore.ieee.org/document/5009015 (visited on 08/25/2024).

[37] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, “Online System Problem Detection by Mining Patterns of Console Logs,” Dec. 2009, pp. 588–597. doi: 10.1109/ICDM.2009.19.

[38] W. Xu, “System Problem Detection by Mining Console Logs,” en, 2010.