# LazyMiner: Project Report

Patrick Hammer
Affiliation: Computer & Information Sciences
347 SERC, 1925 N. 12th Street
Philadelphia, PA 19122
Email: patham9@gmail.com

***Abstract*—Combining rule mining with inference can support energy-efficient and context-aware applications. In general, such technology opens the door to all sorts of applications that need to extract and work with relations between events. Mining user habits, generating efficient sensing plans, sending context-relevant notifications, these and a lot more options arise from that. Systems such as ACE (see [1]) and MobileMiner (see [2]) allow for rule mining, but due to high computational costs and resulting energy demands, the rule mining process is usually done on a remote server rather than on mobile devices directly. We address this issue, by applying a Non-Axiomatic Reasoning System (see [3], [4]) to this problem. Since these systems operate under the Assumption of Insufficient Knowledge and Resources (AIKR), LazyMiner is able to operate under hard resource constraints. This makes it viable to run it directly on the smart-phone. The amount of processing steps per time and per event input, as well as the available memory are exactly configurable. Furthermore, Attentional Control guides the system towards finding stable patterns in floods of events, even when only few resources are available.**

CONTENTS



## I. INTRODUCTION

Rule mining plus inference allows for enhancements of existing applications but also for new possibilities: learned and inferred knowledge can reduce sensing demands and lower energy consumption, can give insights about user habits, and can be used to provide users with contextually relevant information. Such technology can support and complement a large variety of energy-efficient and context-aware applications. It opens the door to all sorts of applications that need to extract and work with relationships between events. However, allowing for efficient rule mining directly on mobile devices is a challenge. Solutions such as [1] and [2] circumvent that problem by letting the rule mining process happen on a server. The LazyMiner is an attempt to get rid of this restriction by allowing for incremental rule mining with user-defined resource usage. LazyMiner makes use of a Non-Axiomatic Reasoning System (NARS) [3] that is designed to work under the Assumption of Insufficient Knowledge and Resources (AIKR). Applying this system allows for precise control of how many

resources should be assigned to the rule mining process, making it tractable to be applied directly on mobile devices. Here, the rule mining happens through the application of temporal inference rules, which allow to summarize temporal patterns in event streams. In such event streams, the amount of possible event combinations grows exponentially with the amount of input events. Thus, exhaustively evaluating all possible combinations is extremely expensive and often not an option. Instead, the Attentional Control mechanism of NARS is utilized, which allows to allocate resources on patterns that seem promising, stable and useful, while keeping the overall resource usage of the system constant. The last chapter shows evaluation results of the effectiveness of LazyMiner when compared to exhaustive rule mining approaches, giving insights of when LazyMiner can be applied and when it shouldn't be used.

## II. Background and comparison

There is related work in the literature, such as MobileMiner, which is described in [2]. MobileMiner also supports rule mining, but also prediction based on rule mining results. However, their approach suffers from a key limitation: the rule mining happens in an exhaustive and non-incremental manner, meaning that the resource usage depends exponentially on the amount of input. To reduce resource usage, they make use of the "Weighted Mining of Temporal Patterns" (WeMit) idea. This idea consists of grouping events within a certain chosen time window into the same baskets, and to represent re-occurring baskets as a single compressed basket. This treatment reduces the amount of possible basket combinations significantly, for all the cases where it applies. But even with this idea, their algorithm scales exponentially in respect to the amount of resulting compressed baskets. In the worst case of temporally sufficiently far apart and distinct input events, the amount of resulting baskets is equal to the amount of input events itself. Thus, for certain scenarios, the rule mining easily gets too expensive to be carried out on mobile devices directly. LazyMiner on the other hand mines patterns incrementally and attention-driven instead, while keeping the total resource usage within a pre-defined constant. Another related work is "The Acquisitional Context Engine" (ACE) that is described in [1]. Here, rule mining works in an incremental manner, and is targeted at energy-efficient context sensing. It also allows for boolean inference based on the rule mining results. While for this approach it is less clear whether it could be made efficient enough to run on a smart-phone, its limitations to only include minConf=99% results for its reasoning, makes it inapplicable for many cases: it misses rules that most time apply, but not always. And by this it also misses potential predictions and inference results, making it effectively fail in less stable environments. For LazyMiner the situation is different since it makes use of Non-Axiomatic Logic (NAL) (see [3]) that allows for conclusions with variable certainty, dependent on the certainty of the premises.

## III. Description

### A. Overview

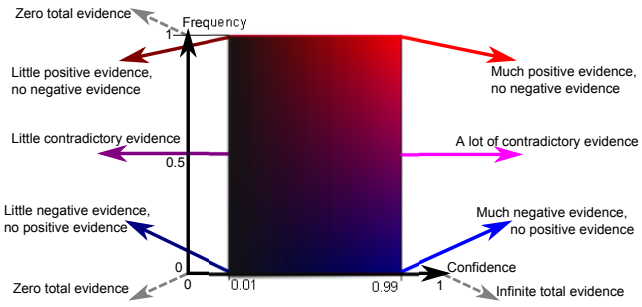
LazyMiner addresses the issue of usually high computational expense in rule mining processes on mobile devices. Having NARS as core component, LazyMiner allows to define the maximum computational resource usage in before-hand through an API call. This makes it tractable to be used on mobile devices. One goal of the project is to show that most useful results are found by the Attentional Control mechanism, even when low resources are given and a high amount of input events is put into. This is demonstrated by cases where often re-appearing patterns are found while operating under such conditions. Such cases consist of re-occurring sequences of events, with potential other events in-between. Also reasoning and question answering capabilities are shown based on such examples. Here, the events are usually incorporating data from all sorts of sensory devices. Being able to use sensory input directly is an additional goal of the LazyMiner project. Several input encoders that map typical sensory data to statements in Non-Axiomatic Logic (NAL) were developed for this purpose.

### B. Architecture

*1) Setup:* As demonstration hardware, a LG Arist MS210 was used, with Android 7.0 as operating system. LazyMiner runs with Android 4.4 (using the ART runtime instead of Dalvik) and newer. As NARS implementation, OpenNARS [4], [5] is integrated. This implementation was chosen as it provides the most complete implementation of NAL. OpenNARS is instantiated a single time and is controlled by LazyMiner. LazyMiner comes with a convenience API and a variety of encoders that allow for an easy way to feed in all sorts of sensory events into the NARS instance. The encoders automatically map sensory data to the format the NARS

instance accepts. Also the LazyMiner API supports registering callback functions for found answers to questions, procedures to define the allowed resource-usage, as well as methods for retrieving inference results. Since LazyMiner is written in Java, using the LazyMiner API from Android applications is straightforward. Besides this, a basic graphical user interface is incorporated that allows for monitoring the state of the NARS instance. Using the GUI, the reasoner can be interactively watched. This makes it easy for application designers to decide on the resources they want to provide the NARS instance with. But it also makes it easy to monitor and query the NARS instance for other issues.

*2) Term Logic:* Since LazyMiner makes use of NARS in the background, all its input is translated into a term logic. This term logic usually consists of $(A \rightarrow B)$ statements which represents an Inheritance relation between A and B. Another relation is Implication. Implication statements are typical result of Temporal Induction, a type of inference that supports rule mining, with results such as $(A \nRightarrow B)$ where both $A$ and $B$ themselves are usually statements/events. Here $\nRightarrow$ is taken as a correlative rather than causal relation, see [6] for why this is the case. The truth value of such statements is however not boolean, it is a tuple of the Positive Evidence $w_+$, and the Negative Evidence $w_-$ that votes for/against the statement. Additionally, the Total Evidence of a statement is intuitively defined as $w := w_+ + w_-$. An alternative representation we will use from now on is defined as the tuple $(f, c)$ with frequency $f := \frac{w_+}{w}$ and confidence $c := \frac{w}{w+1}$. This representation can be mapped to the former more intuitive representation in a bi-directional manner, the following picture illustrates this:




To form such statements, the system can form compound events such as sequences of events. In sequences, the relative time information between the involved events is retained. Additionally, Temporal induction, an Induction rule, allows the system to create predictive statements based on two events. The induction rule basically states[1]

$$A, B \vdash (A \nRightarrow B)$$

and the result allows $A$ to predict $B$ by Deduction. In simplest case, $A$ and $B$ are events containing an Inheritance statement by themselves.

*3) Sensor value encoding:* The simplest way of encoding values into terms is to make use of discretization. Using this method, a term is assigned to a certain numeric range. This makes it possible to the system to explicitly reason about the involved input quantities. The result of such a quantization is usually an event like

```
<{50} --> heartrate>.
```

This is the representation[2] the default sensory value encoders in LazyMiner use. Additionally there is a mode for preserving the numeric quantities completely by modulating the truth value (frequency) of the statement. This idea works when no terms for intermediate values are necessary. This isn't the case in general, but for instance it makes sense to encode the brightness value of a pixel as a statement in such a way that its truth frequency corresponds to its brightness. This leads to a representation like

```
<{pixel1} --> [bright]>. %degree%
```

where degree is the truth frequency of the statement. The distinction between both methods is related to brightness vs. color vision in human beings, a topic that is beyond the scope of this report.

*4) Attentional Control:* In this report, only the for LazyMiner most relevant aspects of the control mechanism of NARS are described. More details can be found in [4]. Attentional Control is achieved by using the Bag structure. This is a data structure where the elements are sorted according to their priority, and the sampling operation chooses candidates with selection chance proportional to their priority. This makes the control strategy similar to the Parallel Terraced Scan in [7], as it also allows to explore many possible options in parallel, with more computation devoted to options which are identified as being more promising. After the selection, a candidate is put back into the bag, but with by a factor (durability) decreased priority value. Data

[1] With the truth value calculation omitted, see [3] for more details.

[2] As the notation suggests, it is the ASCII version of $(\{50\} \rightarrow heartrate)$

items enter with having their priority modulated by their truth value and occurrence time. This is the case for both, input events and results generated by the inference process. All combinations of premises happen through sampling from a bag. Both recent items and high truth-value ones tend to be used more often, but also query-related ones. This suffices to allow for simple attentional control, which chooses premises neither fully randomly nor exhaustively, but biased by the relevance of patterns and the current time. This is helpful as each inference step is valuable: only a fixed amount of them can happen per time unit, so how to spend them matters a lot.

## IV. Reached outcomes

### A. Overview

At first, OpenNARS was ported to Android in a way that it compiled. Porting the Java8 code to the Android platform only required minor changes to the code.

Then, the API was refined that made it possible to feed NAL statements into the reasoner and to register callback functions for reasoning results. Also, a simple user interface was developed that allows to monitor the reasoner using that API.

Also, a LazyMiner-specific API was added, as application designers do not want to deal with the details of NARS when applying LazyMiner, this API acts as an abstraction layer. It encodes a large range of device-specific events and other custom events in a format NARS accepts, for instance it can deal with numeric array inputs. Additionally it allows to define the resource usage. And the developed GUI allows for convenient usage and monitoring in general.

Then, an interactive application in the domain of health care was developed. It shows different abilities of LazyMiner, such as to learn from sensory data and other events, as well as to answer questions about learned knowledge. It does so based on simulated scenarios. In one case, the heart rate of a simulated patient raises unexpectedly and LazyMiner has to answer why.

Last but not least, an evaluation of LazyMiner was attempted that compares it with exhaustive rule mining.

### B. Patient example

In this simulation, there is a heart rate sensor that constantly inputs numeric events like

```
<{60} --> heartrate>.
```

into the system. Additionally, location information and activity information such as whether the person is running, sleeping or walking is provided to LazyMiner. Location information is encoded as an

```
<(*,{SELF},location) --> at>.
```

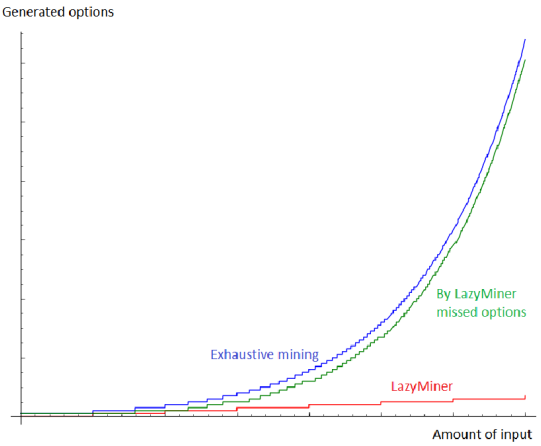
event, and activities are encoded by operations like

```
<(*,{SELF}) --> ^run>.
```

Using these events, the system is queried for why the heart rate reaches a frequency of 110. In the simulation, this value is only reached after the patient is eating at home, potentially because something poisonous was eaten. In the experiments, LazyMiner showed no issues in finding “eating at home” as an explanation of the increased heart-rate after observing this case.

### C. Evaluation

As a baseline for the evaluation, the performance of the system was compared with an exhaustive rule miner. It provides an upper bound to compare with, as by definition there is no way for LazyMiner to find a pattern that is not also found by an exhaustive approach. So probably the ratio between patterns found by LazyMiner divided through the amount of total possible gives an indication of overall capability under the chosen resources? What we obtained is the following, using an input stream of a fixed amount of input events per time unit:




(Particular axis values were omitted) What the picture suggests is that the options missed by LazyMiner grow exponentially in respect to the amount of input events seen so far, meaning the system misses most of the options.

The red line can be easily explained, as the amount of processing LazyMiner can do from one time unit to the next is a constant. So the amount of new patterns it will derive will at most linearly increase with the amount of input / time units. (less than linear when combinations that were already found are generated again)

For the blue curve it is easy to see why it is really growing exponentially: for a list of events $a_1, ..., a_n$ and a new incoming event $a_{n+1}$, at least the options $a_1, ..., a_n$ and $(a_1, a_{n+1}), ..., (a_n, a_{n+1})$ are both possible, so there are at least twice as many candidate combinations of events after a new event is inserted.

Putting these considerations together, with the green function being the red at most linear function subtracted from the blue exponential one, we indeed end up with an exponential function.

And how could it have been otherwise? The resource effort to generate all options grows exponentially with each new incoming event, while LazyMiner invests the same constant amount of time on event 3 as it does on event one million. Even when more resources are assigned to LazyMiner, the slope of the red curve will be steeper, but the result will be the same.

Does this imply low capability of LazyMiner? We claim no. Good explanations usually should be short and communicable and should only contain a small amount of past events rather than most of them. They key is to see that there is a difference between missing most results and missing the most relevant results. As long as LazyMiner does not have the latter issue, it is fine, and will allow for use cases with streams of input events, leading to an extremely scalable approach. But what defines the relevance of a result, when is an event important to consider to be combined with another?

There are different criteria:

1) Is the event related to a question of the user? (is it query-related)
2) Is the past event temporally near to a current candidate to combine with? (did it happen recently)
3) Did the same event happen often in a similar context? (does it correspond to a strong pattern)

The list here is non-exhaustive, but shows cognitively plausible criteria. It is usually much more difficult to see a connection when the events happen temporally far apart from each other, due to attention- and forgetting-related considerations. The same is true for LazyMiner, due to its Bag-based control mechanism. Also it is much easier to see a pattern when it is related to a desired (or wondered about) outcome as Classical Conditioning experiments indicate (see [8]. In LazyMiner a query usually involves a consequence like in $(?how \Rightarrow cons)$. When such a query is issued, the events that have $cons$ as term will gain higher priority, making query-relevant patterns easier to see to LazyMiner. Additionally, it is easier to see a pattern when it repeats often, which again can be explained with the control strategy of LazyMiner. Control strategies that exhibit this property can be easily justified, as the strongest patterns in rule mining are usually by definition the most often occurring ones, so missing patterns that occur rarely usually have little effect.

To complete the evaluation, we use the following setup to show that LazyMiner can indeed be capable for finding the most stable and useful patterns, even though it misses to find most possible patterns:

The input is chosen in such a way that expected to be seen correlations between events are represented by repeating event sequences where the events are not too far apart from each other in time. For such scenarios in our tests, the strongest found patterns were often the same in the exhaustive approach and LazyMiner. In one of these experiments, we used a sequence of letters “abcdabxd” that was encoded in an OpenNARS compatible way as:

```
<{a} --> [observed]>. :|:
10
<{b} --> [observed]>. :|:
10
<{c} --> [observed]>. :|:
10
<{d} --> [observed]>. :|:
10
<{a} --> [observed]>. :|:
10
<{b} --> [observed]>. :|:
10
<{x} --> [observed]>. :|:
10
<{d} --> [observed]>. :|:
```

LazyMiner easily sees the $ab$ pattern. On the other hand it will rarely relate the first $a$ with the last $d$ when low resources are available. Instead it tends to combine events that are more close to each other. This also helps in recognizing the $ab$ pattern again when it occurs again, giving further evidence for $(a \Rightarrow b)$, while there is less

evidence for $(c \nRightarrow d)$ in this example. We evaluated whether this is indeed the case by querying the system for the evidence for these two statements. This was encoded as:

```
//After a, will b occur?
<(&/,<{a} --> [observed]>,?distance) =/>
     <{b} --> [observed]>>?
//After c, will d occur?
<(&/,<{c} --> [observed]>,?distance) =/>
     <{d} --> [observed]>>?
```

with the answers of the system being:

```
<(&/,<{a} --> [observed]>,+12) =/>
     <{b} --> [observed]>>. %1.00;0.38%
<(&/,<{c} --> [observed]>,+12) =/>
     <{d} --> [observed]>>. %1.00;0.31%
```

As expected, with a confidence of 0.38, more evidence for $(a \nRightarrow b)$ was found than with 0.31 for $(c \nRightarrow d)$. The exhaustive miner would usually agree here and in general would often agree in similar examples where the above criteria are mostly met. But it would evaluate much more options than LazyMiner, so would waste resources in such scenarios. One strength of LazyMiner clearly is that events can stream into the system all the time. The system won't get unresponsive to new input, and it mines for patterns in an incremental manner.

## V. API

This sextion gives an overview of how to use the system in applications by using its API.

### A. Input functions

AddSensorEvent allows to enter a certain event with potential associated value(s).

```
AddSensorEvent(String name, int[] value)
AddSensorEvent(int ID, int[] value)
AddSensorEvent(int ID, int value)
AddSensorEvent(String name, int value)
AddSensorEvent(int ID)
```

For instance AddSensorEvent("heartrate",50); will lead to an input event that encodes the current heart-rate of 50.

Additionally, using

```
setDiscretization(int val)
int getDiscretization(int val)
```

can be used to define or get the numeric accuracy for the discretization of values to terms, which can be useful in case that the default accuracy is too low for a certain use case.

Activities that are currently done can be added using the

```
AddActivity(String name)
```

method. Additionally

```
AddAttributeEvent(String instance,
       int value, String attribute)
AddAttributeEvent(String instance,
    String value, String attribute)
AddPropertyEvent(String instance,
                 String property)
```

can be used to define properties and attributes. For instance, AddAttributeEvent("SELF", "home", "at") can encode an location event that says that the phone is currently at home.

### B. Issuing queries

Last but not least, there are functions to query the system for an answer, such as:

```
HowSensorReachedValue(int ID, int value)
HowSensorReachedValue(String name, int value)
HowAttributeReachedValue(String name, int value)
HowPropertyWasFulfilled(String name)
```

These are non-blocking procedures, the system calls back on

```
void event(Class event, Object[] args)
```

using the EventEmitter interface when it found an answer to the query. Then, the event will be Events.Answer.class and the args will include the ID/name to refer to as well as the belief that answers the query.

### C. Configuration

Configuration can be achieved in the constructor of LazyMiner. Here one has full access to the Parameters field, which allows to configure different system parameters, such as bag sizes and inference speed.

## VI. Conclusion

LazyMiner makes rule mining with constant computational resources possible. While it is not the ideal choice when all possibilities need to be evaluated, it allows for energy-efficient open-ended rule mining on mobile devices, and can support context-aware applications that can benefit from its usage. Furthermore, our demonstration example shows that the system is capable of finding answers to queries, such as to find reasonable explanations for an event of interest. Thus, the project goals are met, but LazyMiner and in general applying Non-Axiomatic Reasoning systems to such tasks will be further explored in the future. Also the code of the project will be available on Github, see [9].

## Acknowledgements

Special thanks to Dr. Pei Wang, for his suggestions of how NARS could be applied in this domain.

## References

[1] N. Suman, "Ace: exploiting correlation for energy-efficient and continuous context sensing," *Proceedings of the 10th international conference on Mobile systems*, pp. 29–42, 2012.

[2] S. Vijay, S. Moghaddam, A. Mukherji, K. K. Rachuri, C. Xu, and E. M. Tapia, "Mobileminer: Mining your frequent patterns on your phone," *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pp. 389–400, 2014.

[3] P. Wang, "Non-axiomatic logic: A model of intelligent reasoning," *World Scientific*, 2013.

[4] P. Hammer, T. Lofthouse, and P. Wang, "The opennars implementation of the non-axiomatic reasoning system," *Springer International Publishing*, 2016.

[5] "Opennars repository." [Online]. Available: https://github.com/opennars/opennars

[6] P. Wang and P. Hammer, "Issues in temporal and causal inference," *Springer, Cham*, 2015.

[7] J. Rehling and D. Hofstadter, "The parallel terraced scan: An optimization for an agent-oriented architecture," *Intelligent Processing Systems*, 1997.

[8] I. Pavlov, "Conditioned reflexes: An investigation of the physiological activity of the cerebral cortex," *Oxford University Press*, p. 142, 1927.

[9] "Lazyminer repository." [Online]. Available: https://github.com/patham9/lazyminer