SoftwareX 31 (2025) 102341

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

PyAerial: Scalable association rule mining from tabular data

Erkan Karabulut *, Paul Groth, Victoria Degeler

University of Amsterdam, 1098 XH, North Holland, The Netherlands

A R T I C L E   I N F O

Keywords:
Association rule mining
Neurosymbolic ai
Tabular data
Interpretable machine learning

A B S T R A C T

Association Rule Mining (ARM) is a knowledge discovery technique that identifies frequent patterns as logical implications within transaction datasets and has been applied across domains such as e-commerce, healthcare, and cyber–physical systems. However, many state-of-the-art ARM methods, typically algorithmic or nature-inspired, suffer from rule explosion and long execution times. Aerial is a novel neurosymbolic ARM algorithm for tabular datasets that mitigates rule explosion using neural networks, while remaining compatible with existing approaches. Aerial transforms tables into transactions, uses an autoencoder to learn compact neural representations, and extracts logical rules from the neural representations. This paper presents PyAerial, a Python library that makes Aerial accessible and easy to use on generic tabular datasets for end users in a domain-independent way. Besides association rules, PyAerial can also be used to extract frequent itemsets, learn classification rules, apply item constraints to learn rules over the features of interest rather than all features, pre-discretize numerical data for ARM, and can be run on a GPU.

Code metadata

Current code version	v1.0.3
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-25-00494
Permanent link to Reproducible Capsule	https://codeocean.com/capsule/2772444/tree/v1
Legal Code License	MIT License
Code versioning system used	Git
Software code languages, tools, and services used	Python
Compilation requirements, operating environments & dependencies	Python 3.8 or above
If available Link to developer documentation/manual	https://github.com/DiTEC-project/pyaerial
Support email for questions	GitHub issues (https://github.com/DiTEC-project/pyaerial/issues)

1. Motivation and significance

Association Rule Mining (ARM) is the knowledge discovery task of mining patterns in transaction databases — structured collections where each entry records a set of items involved in a transaction — in the form of logical implications [1], typically denoted as $\mathcal{A} \rightarrow \mathcal{C}$, meaning 'if $\mathcal{A}$ then $\mathcal{C}$'. ARM has applications in a myriad of domains, including healthcare [2], e-commerce [3], and cyber–physical systems [4]. The traditional ARM algorithms mostly include algorithmic [5] and optimization-based [6] methods, which often result in a high number of rules and long execution times, known as the rule explosion problem. In addition to knowledge discovery, ARM is utilized in downstream classification tasks as part of interpretable Machine Learning (ML) models for high-stakes decision-making [7], and the rule explosion problem propagates to the downstream tasks as processing a high number of rules is resource-intensive.

ARM is inherently a combinatorial problem due to the exponential search space of possible itemsets satisfying given rule quality criteria, causing rule explosion. Many combinatorial problems have recently been addressed using neural networks [8,9]. A state-of-the-art Neurosymbolic ARM approach (Aerial) has been recently introduced to tackle the rule explosion problem using neural networks for both Internet of Things (IoT) data [4,10] and generic tabular datasets [11] represented as transaction databases. Aerial has two steps: (i) it creates a compact neural representation of the data using an under-complete denoising Autoencoder [12], (ii) then extracts association rules from

* Corresponding author.
E-mail addresses: e.karabulut@uva.nl (Erkan Karabulut), p.t.groth@uva.nl (Paul Groth), v.o.degeler@uva.nl (Victoria Degeler).

https://doi.org/10.1016/j.softx.2025.102341
Received 12 July 2025; Received in revised form 18 August 2025; Accepted 28 August 2025
Available online 13 September 2025

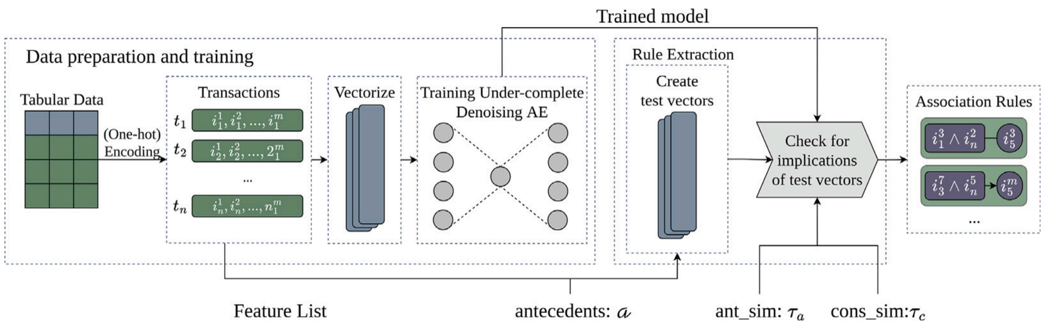

Fig. 1. Rule mining pipeline of Aerial consists of data preparation, training, and rule extraction steps [11].

the neural representation using a novel rule extraction algorithm. In this way, Aerial can learn a concise set of high-quality association rules with full data coverage and facilitate downstream classification tasks significantly in terms of execution time and accuracy. The runtime analysis of Aerial in [4,11] shows that it is scalable on large datasets. Furthermore, Aerial is compatible with the existing work on tackling rule explosion, such as ARM with item constraints [13–15], top-k rule mining [16], and closed-itemset mining [17].

Contributions. This paper presents PyAerial (see Code metadata table), a scalable and user-friendly Python library that brings the Aerial framework to generic tabular datasets. Main contributions:

- **Ease of use:** PyAerial makes the Aerial framework accessible to end users with just two lines of code, enabling domain-independent application on any tabular dataset.
- **Customizability:** The library allows users to fine-tune Aerial's underlying Autoencoder architecture as well as the rule extraction process via simple, flexible parameter settings.
- **Comprehensive functionality:** PyAerial offers a rich set of features:
  - Mining frequent itemsets
  - Extracting association rules
  - Applying item constraints to both the antecedent and consequent of rules, allowing targeted rule learning over selected features
  - Discovering classification rules with a class label as the rule consequent
  - Pre-discretizing numerical features for association rule mining (ARM)
  - Computing rule quality metrics such as support, confidence, coverage, and Zhang's metric [18]
  - Leveraging GPU acceleration for improved performance

2. Software description

This section provides a brief overview of Aerial to facilitate an understanding of the software architecture of PyAerial. For a detailed description and extensive evaluation of Aerial, please refer to [11].

Aerial ARM pipeline for tabular datasets consists of three main steps as shown in Fig. 1: (i) data preparation, (ii) training, and (iii) rule extraction.

- **Data preparation.** Tabular data is converted to transactions via one-hot encoding. Each transaction contains items that indicate the presence or absence of a certain value for each column of the table. Transactions are then converted to vector form for neural network training.
- **Training.** Aerial utilizes an under-complete denoising Autoencoder [19] to learn a compact representation of the tabular data. The Autoencoder learns to reconstruct each row of the table using the compact representation, and outputs a probability distribution per column, e.g., one probability value for each of the possible values in a column. In doing so, it learns the associations between columns of a table.
- **Rule extraction.** Aerial exploits the reconstruction feature of its Autoencoder architecture to extract association rules. After training, if a forward run on the trained model with a set of marked (column) categories A results in successful reconstruction (high probability) of categories C, we say that marked features A imply the successfully reconstructed features C, such that $A \rightarrow C \setminus A$ (no self-implication).

As shown in [4,11], Aerial results in a more concise number of high statistical quality rules with full data coverage than the state-of-the-art approaches. Existing work on tackling the rule explosion problem can be incorporated into Aerial as exemplified in both [11] and implemented as part of the PyAerial Python package as described in Section 2.2.

Therefore, Aerial can be used in any domain with tabular data structures, both for knowledge discovery as well as downstream classification tasks for interpretable and high-stakes decision-making [7].

2.1. *Software architecture*

Fig. 2 shows the software architecture of PyAerial. The arrows going out from the user's representation correspond to the modules that the users have access to. PyAerial consists of 5 main modules: (i) model, an under-complete denoising Autoencoder implementation using PyTorch [20], (ii) rule extraction module that extracts both frequent itemsets and association rules of different form, (iii) discretization module that provides basic numerical data discretization method for ARM, (iv) rule quality module that provides the most common association rule quality metrics, and (v) data preparation module which the PyAerial itself uses for preparing the data for ARM.

Basic flow. PyAerial creates an under-complete denoising Autoencoder with a certain number of layers and dimensions that is decided automatically based on the input table dimensions. This is explained in detail in Section 2.2.1. Users can also choose to alter the Autoencoder architecture by changing the parameters of the `train()` function. Before training, the discretization module can be used to auto-identify numerical columns and discretize them for ARM. After training, the rule extraction module can be used to extract both frequent patterns of varying lengths or association rules with a given size and form. Lastly, the rule quality module provides a wide variety of quality metrics that are common in the ARM literature [6]. In the simplest use case of learning rules from a given table without fine-tuning, users can call the `train()` and `generate_rules()` functions to learn association rules in two simple steps.

2.2. *Software functionalities*

This section presents the functionalities of PyAerial.

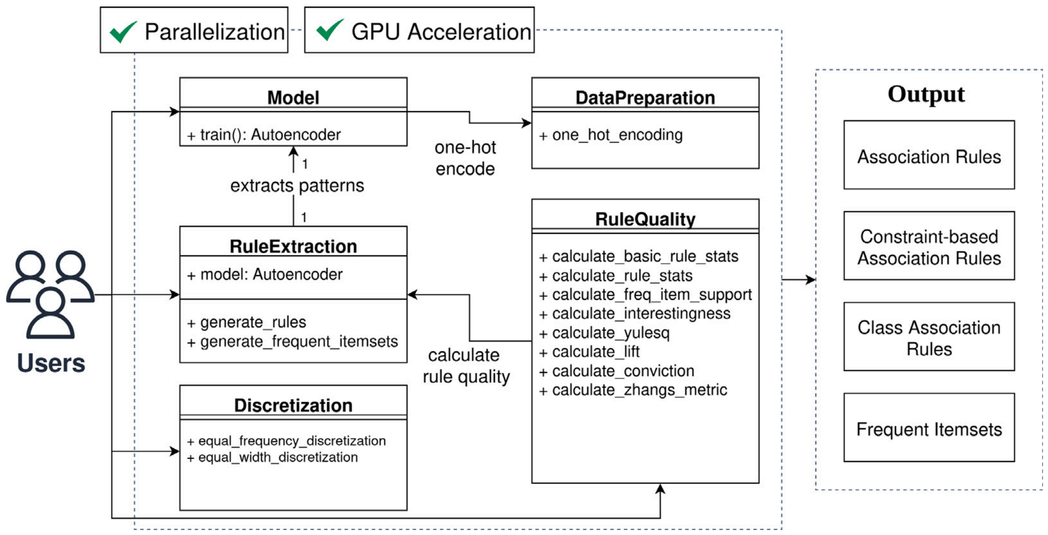

Fig. 2. PyAerial's classes and functionalities as a UML diagram.

2.2.1. Association rule mining from tabular data.

The basic use case of PyAerial is to learn association rules from a given categorical tabular dataset in the form $\mathcal{A} \rightarrow \mathcal{C}$, where $\mathcal{A}$ is a set of items (value of a certain column) and $\mathcal{C}$ is a single item. This is done in two steps:

- **Training.** Upon calling the `train(transactions:pandas.Dataframe): Autoencoder` function with a tabular dataset in a 2d pandas dataframe form [21], PyAerial first initializes an Autoencoder (given under the `model` module) with an auto-selected number of layers and layer dimensions based on the table characteristics. This selection is derived from our extensive experiments in papers [4,11], however, fine-tuning of the hyperparameters may be required depending on dataset characteristics and application-specific goals of rule learning, which is described in detail in Section 2.2.6. By default, the number of layers is chosen based on the logarithm of the input size (base 16), and the hidden layer sizes are computed by gradually shrinking the input dimension down to the output size using a constant reduction ratio. This creates a symmetrical encoder–decoder architecture that compresses and reconstructs the data. PyAerial then one-hot encodes the given table, and performs an Autoencoder training with reconstruction success as the loss function, having probabilities per column in the output layer. It then returns the trained Autoencoder instance.
  In addition, users can also specify the number of layers and the dimension of each layer for fine-tuning. The following is a full list of parameters that can be adjusted by the users:
  - `transactions` (`pd.DataFrame`, required): Input tabular data to be used for training the autoencoder.
  - `autoencoder` (`AutoEncoder`, optional): Custom autoencoder instance; if not provided, a new one is created.
  - `noise_factor` (`float`, optional): Amount of noise added to the input data for denoising, default is 0.5.
  - `lr` (`float`, optional): Learning rate for the optimizer, default is 5e-3.
  - `epochs` (`int`, optional): Number of training epochs, default is 1.
  - `batch_size` (`int`, optional): Number of samples per batch, default is 2.
  - `num_workers` (`int`, optional): Number of worker threads for data loading, default is 1.
  - `layer_dims` (`list`, optional): Custom list of hidden layer dimensions; if not provided, defaults are computed automatically.
  - `device` (`str`, optional): Computation device to use (e.g., `‘‘cuda’’` or `‘‘cpu’’`); selected automatically if not specified.
- **Rule extraction.** The second step is to extract rules from the trained Autoencoder by calling the `generate_rules(trained_autoencoder): list` function of the *rule extraction* module. This function uses Algorithm 1 of [11]. During training, the Autoencoder learns co-occurrence probabilities of column values as part of its compact neural representation. The rules are then extracted based on their antecedent and consequent co-occurrence probabilities. As a result, this function returns a list of association rules.
  The following is the list of parameters of `generate_rules(...)`:
  - `autoencoder` (`AutoEncoder`, required): A trained autoencoder used to extract association rules.
  - `features_of_interest` (`list`, optional): A list of features or feature-value pairs to constrain rule antecedents; format includes strings or dictionaries like `{‘‘feature’’: ‘‘value’’}`. A string representing a column name will result in including all values of that column in the rule extraction, while a dictionary of feature-value pairs will only result in including the given values for that feature (column name) in the extraction.
  - `ant_similarity` (`float`, optional): Minimum similarity threshold for selecting antecedents; default is 0.5.
  - `cons_similarity` (`float`, optional): Minimum similarity threshold for selecting consequents; default is 0.8.
  - `max_antecedents` (`int`, optional): Maximum number of features allowed in a rule's antecedent; default is 2.
  - `target_classes` (`list`, optional): A list of features or feature-values that should appear on the consequent side of the rules. This parameter has the same format as the `features_of_interest` parameter.

A **guideline** on selecting the **hyperparameters** is given in Section 2.2.6.

Table 1
Sample entries from the mushroom dataset in [22].

cap-shape	cap-surface	cap-color	bruises	odor	...	poisonous
b	y	w	t	l	...	e
x	y	n	t	p	...	p
b	s	y	t	l	...	e

2.2.2. *Frequent itemset mining from tabular data*

Another major feature of PyAerial is the ability to extract frequently occurring patterns, frequent itemsets, from the trained Autoencoder. Similar to learning association rules, this is done in two steps: training and frequent pattern extraction. The training step is the same as in Section 2.2.1. Frequent itemset extraction is done using the `generate_frequent_itemsets (trained_autoencoder: Autoencoder): list` function. This function uses Algorithm 2 of [11], and the extraction logic is similar to extracting association rules. Given a set of marked itemsets I_1, if the trained autoencoder successfully reconstructs a set of itemsets I_2 based on a given reconstruction probability threshold τ_i, we say that I_1 and I_2 are frequent itemsets. Parameters are as follows:

- `autoencoder (AutoEncoder`, required): A trained autoencoder used to extract frequent itemsets.
- `features_of_interest (list`, optional): A list of features or feature-value pairs to constrain frequent itemsets. This parameter has the same format as the `generate_rules(...)` function.
- `similarity (float`, optional): Minimum frequent itemset similarity threshold; default is 0.5.
- `max_length (int`, optional): Maximum number of items in each frequent itemset; default is 2.

2.2.3. *ARM with item constraints*

In some scenarios, users are primarily interested in associations involving a subset of features rather than the entire feature set [13]. This task, known as ARM with item constraints, focuses rule discovery on selected features, which helps streamline post-processing and mitigates the rule explosion problem.

PyAerial supports this functionality through the `features_of_interest` optional parameter in the `train(...)` function, as described earlier. Users can specify features of interest either by listing specific feature-value pairs or by selecting entire features (including all their possible values). PyAerial then restricts rule extraction to the specified features of interest.

2.2.4. *PyAerial on numerical values*

ARM methods were initially tailored for categorical data [5], which then also extended into numerical features [6] either by pre-discretization or using optimization-based methods. PyAerial can auto-identify numerical values and offers two of the common pre-discretization methods as part of the `discretization` module, equal-frequency and equal-width discretization [23].

`equal_frequency_discretization (transactions: pandas.Dataframe, bins: int): pandas.Dataframe` function of the `discretization` module identifies the numerical columns in the given transactions, splits them into a number of `bins`, ensuring that items in each interval have equal occurrence (frequency) in the data. `equal_width_discretization (transactions: pandas.Dataframe, bins: int): pandas.Dataframe` function splits the numerical features into a given number of bins based on equal interval ranges (width).

2.2.5. *Learning class association rules for interpretable inference*

Besides knowledge discovery, ARM is also used as part of interpretable machine learning models, such as rule-based classifiers [24], for high-stakes decision-making [7]. These models build classifiers from a given set of association rules that have a class label on the right-hand side, also known as class association rules [25].

PyAerial supports this functionality via the `target_classes` parameter of the `generate_rules(...)` function. Users can specify the target class (feature) name(s) using this parameter. PyAerial then restricts the rule extraction to rules that have the given class target class(es) in the right-hand side.

2.2.6. *A guideline on debugging PyAerial and selecting the hyperparameters*

This subsection provides a basic guideline for debugging PyAerial and selecting hyperparameters. For an extended version, please refer to the GitHub page.[1] and the paper [11]

What to do when PyAerial does not learn any rules? If no rules are discovered, assuming the input data is properly prepared (e.g., discretized), several strategies may help. Increasing the number of training epochs may allow the model to better capture associations, though it comes with a risk of overfitting. Expanding the model's capacity by increasing the number or dimensionality of layers can also help uncover more complex patterns. Additionally, lowering the `ant_similarity` and `cons_similarity` thresholds — analogous to minimum support and confidence in traditional ARM — can increase the number of discovered rules, albeit with potentially weaker statistical rule quality strength.

What to do when PyAerial takes too much time or learns too many rules? Excessive rule generation or long training times typically indicate a large search space. To mitigate this, one can start with a small search space (e.g., set `max_antecedents=2`) and increase gradually. Use of high values for `ant_similarity` and `cons_similarity` (e.g., 0.5 and 0.9) can further restrict the rule space to only the most prominent associations. Reducing the number of epochs and network parameters can help prevent overfitting, especially in datasets with many rows and few columns, as this can potentially lead to a high number of obvious rules (low association strength). Rule learning can also be focused on specific features of interest using item constraints. For larger models, GPU acceleration is recommended.

Effect of hyperparameters. The `ant_similarity` parameter controls the similarity threshold for antecedents and behaves similarly to a minimum support threshold—higher values yield fewer, higher-quality rules (though not the same). The `cons_similarity` parameter governs the similarity of consequents and corresponds to a joint confidence and interestingness filter—higher values improve rule quality and generalizability. The `max_ antecedents` parameter defines the maximum number of features in a rule's antecedent; increasing this allows more expressive rules but expands the search space. Training-related parameters (e.g., epochs, number of layers, and latent dimensionality) influence both execution time and the model's capacity to represent complex patterns. Shallow architectures may underfit, while overly deep ones risk overfitting and longer runtime. Based on the analysis in [11], for typical datasets with $n \gg d$, a two-layer encoder–decoder with decreasing dimensions and 2–5 epochs usually suffices to achieve meaningful rule discovery.

[1] https://github.com/DiTEC-project/pyaerial#how-to-debug-aerial

3. Illustrative examples

This section provides (i) easy installation steps and example usages of PyAerial and (ii) a performance comparison of PyAerial with prominent ARM libraries.

3.1. Installation and usage

This section illustrates the installation and usage of PyAerial by making use of the mushroom dataset from the University of California, Irvine (UCI) ML repository [22]. This is due to the ease of data loading from the UCI ML repository in Python, so that readers can copy and paste the running code. The dataset contains mushroom features in the columns and mushroom instances in the rows (see Table 1).

PyAerial installation. PyAerial can be simply installed using *pip*:

```
    pip install pyaerial
```

Listing 1 PyAerial can be easily installed via pip.

Learning association rules with PyAerial can be done as follows: (i) load the data, (ii) train the model, (iii) extract association rules:

```
from aerial import model, rule_extraction
from ucimlrepo import fetch_ucirepo

# 1. load the mushroom dataset
mushroom = fetch_ucirepo(id=73).data.
    features

# 2. train an autoencoder on the loaded
    table
trained_autoencoder = model.train(mushroom
    )

# 3. extract association rules from the
    autoencoder
association_rules = rule_extraction.
    generate_rules(trained_autoencoder)
```

Listing 2 Using PyAerial for association rule mining from tabular data.

A sample association rule is shown below:

$\{\text{cap-shape} = x,\ \text{cap-surface} = f\} \rightarrow \{\text{odor} = n\}$

Calculating rule quality metrics such as support, confidence, coverage, and association strength (Zhang's metric [18]) using PyAerial's `rule_quality` module as follows:

```
from aerial import rule_quality
...
# 1. detailed rule quality measurements,
    including overall average statistics
average_stats, association_rules =
    rule_quality.calculate_rule_stats(
    association_rules, trained_autoencoder
    .input_vectors)

# 2. calculate only the basic (support,
    confidence) metrics per rule
association_rules = rule_quality.
    calculate_basic_rule_stats(
    association_rules, trained_autoencoder
    .input_vectors)
```

Listing 3 Calculating association rule quality metrics with PyAerial.

In the first option, the *average_stats* object includes the average rule quality over all the learned rules, while the updated *association_rules* object now includes rule quality metrics for each rule:

$\{\text{cap-shape} = x,\ \text{cap-surface} = f\} \rightarrow \{\text{odor} = n\}$

Support = 0.088,

Confidence = 0.617,

Zhang's metric = 0.346,

Rule coverage = 0.143

The second option only calculates support and confidence for each rule.

Learning class association rules with PyAerial can be done using the `target_classes` option of the `generate_rules(...)` function:

```
...
# Learn rules with "poisonous" feature as
    consequence
association_rules = rule_extraction.
    generate_rules(trained_autoencoder,
    target_classes=["poisonous"])
```

Listing 4 Learning class association rules with PyAerial.

An example class association rule with the "poisonous" label:

$\{\text{cap-surface} = f,\ \text{spore-print-color} = n\} \rightarrow \{\text{poisonous} = e(\texttt{edible})\}$,

confidence = 0.935

Visualizing association rules learned by Aerial is possible through the NiaARM [26,27] library. Please see our GitHub documentation to see how NiaARM can be used together with PyAerial to visualize association rules. One example visualization is given in Fig. 3.

Many more example usages of PyAerial for each of its functionalities can be found on the GitHub page.

3.2. Complementary benchmark

The performance of Aerial+ has been **extensively evaluated in earlier work** based on execution time, rule quality, and downstream classification tasks [4,11]. The goal of this subsection is to provide a complementary benchmark comparing PyAerial with prominent ARM libraries and algorithms.

Prominent ARM libraries. We compare PyAerial with the popular open-source algorithmic and optimization-based ARM libraries for categorical tabular data, which are given in Table 2, as well as another neurosymbolic method, ARM-AE [33]. Note that arulespy only offers a Python interface to the arules [34] library written in R and C, and NiaARM uses NiaPy [35] to execute nature-inspired optimization algorithms for ARM. Other notable libraries include NiaAutoARM [36], which focuses on auto-constructing ARM pipelines using stochastic population-based metaheuristics, and NiaARMTS [37], applying numerical ARM to time series data. However, these libraries are not a direct equivalent of PyAerial as they do not explicitly focus on categorical tabular data, and therefore are excluded from the comparison.

3.2.1. Execution time and number of rules analysis

This section illustrates PyAerial's efficiency on a high-dimensional tabular dataset in terms of execution time and number of rules in comparison to 9 baselines implemented using the libraries from Table 2.

Baselines. The baselines include: Apriori [1] (implemented with arulespy), FP-Growth [38] (MLxtend and SPMF), ECLAT [39] (pyECLAT and arulespy), HMine [40] (MLxtend), ARM implementations of the optimization algorithms Differential Evolution (DE) [41] and Particle Swarm Optimization [42] for categorical data [43] (NiaARM), and

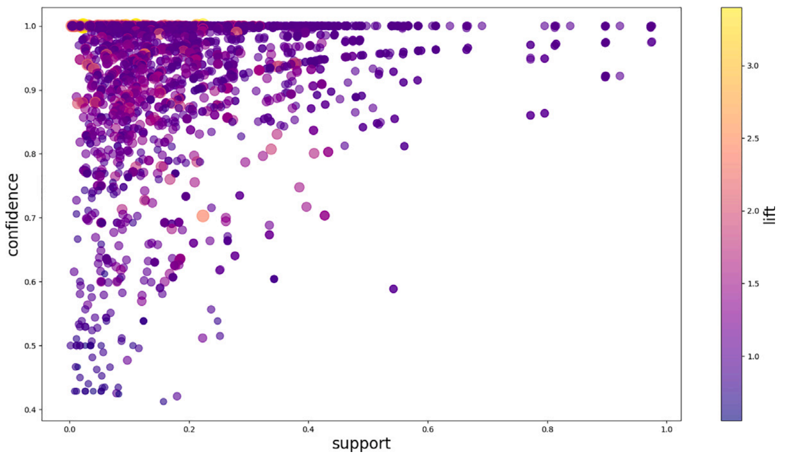

Fig. 3. Visualization of rule quality metrics for PyAerial association rules on the mushroom dataset, generated with the NiaARM library [26,27].

Table 2
Prominent ARM libraries for categorical tabular (or transactional) data.

Library	Language	Type	Notable Features
PyAerial	Python	Neurosymbolic	Uses neural networks to learn prominent features. Supports item constraints, classification rules, GPU execution, and parallelization.
MLxtend [28]	Python	Algorithmic	Supports 4 fundamental ARM algorithms.
SPMF [29]	Java	Algorithmic	Extensive collection of over 260 algorithmic pattern mining methods, including ARM.
pyECLAT [30]	Python	Algorithmic	Simple Python interface for ECLAT.
arulespy [31]	R and C	Algorithmic	Efficient implementation of 7 ARM methods, supports visualizations.
NiaARM [26]	Python	Optimization	Uses nature-inspired algorithms for categorical and numerical ARM, supports visualizations.
PyFIM [32]	C	Algorithmic	Efficient C implementations of over 20 ARM methods, minimal dependencies.

Table 3
A sample gene expression level data from [44]. It is pre-processed and put in a discrete form using z-score binning into 3 bins: low, normal, and high expression levels.

Sample	Gene_1	Gene_2	Gene_3	⋯	Gene_18107	Gene_18107
Sample_1	normal	normal	normal	⋯	normal	normal
Sample_2	normal	normal	high	⋯	normal	high
Sample_3	normal	normal	normal	⋯	normal	low
…						

the neurosymbolic ARM-AE [33] method. We also tested PyFIM's FP-Growth implementation [32]; however, the results it returns are not correct and do not match any other algorithmic method. Therefore, we opted to leave PyFIM out of this evaluation.

Dataset. We use a high-dimensional (18,000+ columns, 86 rows) gene expression levels dataset from the biomedical domain [44]. The dataset is exemplified in Table 3. A pre-processed version of this dataset from [45] is utilized and further pre-processed by applying z-score binning (into low, normal, and high expression levels) to prepare it for categorical ARM.

Setup and hyperparameters. Due to the distinct nature of each baseline (algorithmic, optimization-based, and neurosymbolic), a fair comparison is a challenge. For all the methods with the exception of optimization-based methods (DE and PSO), rules of antecedent length 2 are learned. Note that the antecedent length cannot be controlled for the optimization-based methods. Aerial+ is trained for 25 epochs with all other parameters being the default PyAerial parameters. The minimum support threshold of the algorithmic methods is set to the smallest value covering 90% of the supports of rules learned by PyAerial so that the algorithmic methods will be able to find a majority of the rules that Aerial+ discovers, and the minimum confidence threshold is set to 80%, the same as PyAerial's minimum consequent probability threshold. The optimization-based methods are initialized with a population size of 200 and a maximum evaluation of 50,000, which refer to the initial number of solutions (as these methods represent ARM as an optimization problem) and the number of optimization function evaluations, respectively. The number of rules per consequent parameter of ARM-AE is set to the number of rules found by PyAerial, divided by the unique number of items, so that both algorithms will find similar numbers of rules for fairness, and the number of epochs is set to 25, similar to PyAerial. Default values are chosen for all remaining hyperparameters of all baselines. The **source code** of the experiments can be found under the *illustrative_experiments* folder of the PyAerial repository.

Results. Fig. 4 shows the execution time (includes both training and rule extraction time) and the number of rules for PyAerial and baselines as the number of columns increases. Note that there are 3 unique values per column (low, normal, and high). The results show that PyAerial outperforms all the prominent ARM libraries in terms of execution time as the number of columns increases. Initially, due to the training stage of PyAerial, the algorithmic methods run faster on smaller tables, up until reaching 60 columns (∼180 unique items per transaction). As the number of columns exceeds 60, PyAerial runs up to one to two orders of magnitude faster than the most prominent ARM libraries. Apriori implementation (in C language) of the arules library is the only exception where PyAerial runs only 3 times faster upon reaching 300 columns. On average, PyAerial results in 3 times fewer rules in comparison to algorithmic methods. The optimization-based methods (DE and PSO) could not find any rules after 50 and 60 columns, respectively. This implies that they need longer executions (increasing

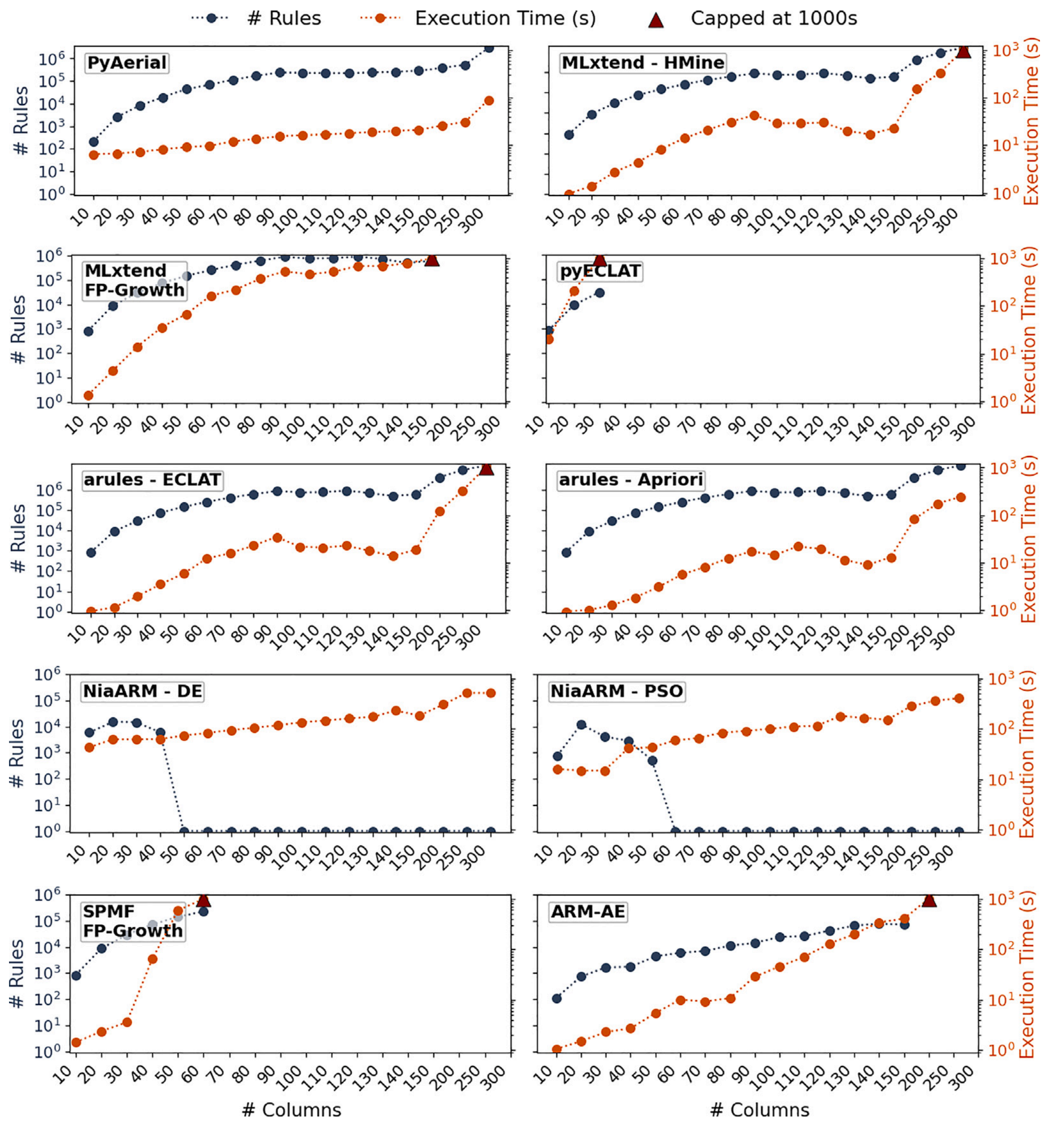

Fig. 4. PyAerial (Aerial+) has one-to-two orders of magnitude faster execution time (includes both training time and rule extraction time) than many prominent ARM libraries, and 3 times faster than the second best method (arules - Apriori). PyAerial also results in the smallest number of rules on average. DE and PSO were not able to find rules after increasing the column count to 60 and 70 (approximately 200 unique items per transaction), needing longer evaluation cycles.

the maximum evaluations parameter) to be able to discover patterns as the number of columns increases. ARM-AE, on the other hand, has already exceeded 10^3 seconds of execution time upon reaching 200 columns, where PyAerial took less than 10^2 seconds for 300 columns.

3.2.2. *Rule quality analysis*

This section further evaluates PyAerial and the baselines in terms of rule quality: average rule support, average rule confidence, and total data coverage of the rules.

Setup and hyperparameters. This experiment uses the exact same setup and hyperparameters as the earlier experiment, with the following exception: Aerial+ and ARM-AE are trained for 2 epochs with 1 hidden layer of dimension 10. This is to find an even more concise number of rules (more prominent patterns). The experiments are run on the first 50 columns of the same dataset.

Results. The results are shown in Table 4. PyAerial, with only 57 rules, reaches full data coverage with the highest confidence across all the prominent ARM libraries. This is thanks to the under-complete Autoencoder architecture of Aerial+, which leads to learning only the most significant rules per feature. We also observe that there are small differences in the implementation of the exhaustive methods. Apriori implementation of arules found slightly more rules than other algorithmic methods, which theoretically should have resulted in the same number of rules. DE could not find any rules and needed longer executions than 50,000 evaluations, while PSO was able to find compatible quality rules to the algorithmic methods.

Further validation of PyAerial. Aerial+ (implemented with PyAerial) is further validated on **many other open-source datasets** in [4,11] in terms of execution time and rule quality. Furthermore, the concise number of rules learned by PyAerial also resulted in higher downstream rule-based classification accuracy and improved execution times significantly when used as part of interpretable ML models such as CORELS [24] (shown in [4]).

4. Impact

In the era of Generative AI, learning association rules remains a relevant research area for two key reasons:

- **Knowledge discovery.** ARM provides a human-understandable and machine-processable approach to knowledge discovery, widely

Table 4
With a significantly more concise number of rules, PyAerial results in the highest confident rules with full data coverage in comparison to the most prominent ARM libraries.

Approach	# Rules	Support	Confidence	Coverage
PyAerial	57	0.53	**0.92**	1.00
ARM-AE	551	0.28	0.37	0.13
MLxtend - FP-Growth	7176	0.53	0.86	1.00
MLxtend - HMine	7176	0.53	0.86	1.00
pyECLAT - ECLAT	7176	0.53	0.86	1.00
arules - ECLAT	7176	0.53	0.86	1.00
arules - Apriori	7286	0.52	0.85	1.00
NiaARM - DE	4	0.02	0.33	1.00
NiaARM - PSO	1335	0.06	0.86	1.00
SPMF - FP-Growth	7269	0.52	0.85	1.00

applied across domains [5,6], and remains valuable despite the rise of black-box deep learning models.

- **Interpretable machine learning.** ARM supports rule-based prediction and classification for high-stakes decisions, such as in medicine and recidivism, due to its interpretable nature [7,24].

Contribution to ARM literature. Aerial tackles the central challenges of rule explosion and long execution time in ARM using neural networks, while staying compatible with existing techniques. It enables scalable ARM on high-dimensional datasets, which previously required search space reduction (e.g., item constraints) and high computational cost.

Evaluation and validation of Aerial. Our prior work evaluated Aerial on IoT [4,10] and generic tabular datasets [11], showing that it outperforms state-of-the-art ARM methods by: (i) producing a more concise set of high-quality rules with full data coverage, (ii) significantly reducing rule learning time, and (iii) improving downstream classification accuracy and efficiency.

PyAerial and its advantages over existing ARM libraries. PyAerial exposes Aerial, a state-of-the-art Neurosymbolic ARM method, through a simple two-line interface (see Section 3) for both learning frequent itemsets as well as association rules, and supports fine-tuning of both its Autoencoder architecture and the rule extraction process. **No other software library** supports any Neurosymbolic methods for ARM. In addition, PyAerial offers advanced ARM features such as item constraints (on both sides of a rule), class association rules (by feature or value), GPU acceleration, and parallelization. Popular Python libraries such as Mlxtend [28] and PyFIM [46] only provide implementations of traditional rule mining methods and lack these listed advanced capabilities. SPMF [47] is an open-source Java data mining library that also includes various ARM methods. However, SPMF also implements only the traditional algorithmic ARM methods and not the state-of-the-art Neurosymbolic methods such as Aerial. Note that besides the aforementioned features of PyAerial, another qualitative advantage is that larger neural network model implementations are often done in Python using, e.g., PyTorch [20], and PyAerial can potentially be integrated into such models for interpretability.

Potential impact of PyAerial. PyAerial applies to any tabular data in a domain-independent way, supporting knowledge discovery and interpretable modeling on high-dimensional datasets. It may also assist in explaining deep neural networks by integrating its rule extraction logic into larger models. Since Aerial was only introduced in 2024, PyAerial's current reach is limited. This paper aims to broaden PyAerial's accessibility across domains.

5. Conclusions

This paper introduced *PyAerial*, a Python library that makes the scalable Aerial Neurosymbolic ARM method accessible to end users with just a few lines of code. Aerial addresses key challenges in ARM, such as rule explosion and long execution times on high-dimensional datasets. PyAerial implements Aerial's capabilities for generic tabular data in a domain-independent way, supporting: (i) frequent itemset mining, (ii) association rule learning, (iii) item constraints on both rule sides, (iv) classification rules with class labels on the right-hand side, (v) pre-discretization of numerical features, (vi) rule quality metrics including support, confidence, coverage, and Zhang's metric [18], and (vii) GPU acceleration. When evaluated in a complementary benchmark of high-dimensional tabular data, PyAerial outperformed 9 prominent ARM library implementations in terms of execution time, rule conciseness, and quality. Thus, PyAerial enables knowledge discovery and interpretable inference on high-dimensional datasets to practitioners.

Furthermore, PyAerial can potentially assist in explaining larger neural networks by integrating its Autoencoder and rule extraction logic into larger models. PyAerial is continuously being improved, e.g., by creating new ARM variants based on Aerial, implementing more rule quality metrics, and integrating into interpretable machine learning models.

CRediT authorship contribution statement

Erkan Karabulut: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Conceptualization. **Paul Groth:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Victoria Degeler:** Writing – review & editing, Supervision, Methodology, Funding acquisition, Conceptualization.

Acknowledgments

This work has received support from the Dutch Research Council (NWO), The Netherlands, in the scope of the Digital Twin for Evolutionary Changes in water networks (DiTEC) project, file number 19454.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Erkan Karabulut reports financial support was provided by Dutch Research Council. Victoria Degeler reports financial support was provided by Dutch Research Council. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Agrawal R, Srikant R, et al. Fast algorithms for mining association rules. In: Proceedings of the 20th international conference on very large data bases, VLDB, vol. 1215, 1994, p. 487–99.
[2] Zhou S, He J, Yang H, Chen D, Zhang R. Big data-driven abnormal behavior detection in healthcare based on association rules. IEEE Access 2020;8:129002–11.
[3] Roy D, Dutta M. A systematic review and research perspective on recommender systems. J Big Data 2022;9(1):59.

[4] Karabulut E, Groth P, Degeler V. Learning semantic association rules from internet of things data. Neurosymbolic Artificial Intelligence 2025. [in press], arXiv:2412.03417.

[5] Luna JM, Fournier-Viger P, Ventura S. Frequent itemset mining: A 25 years review. WIREs Data Min Knowl Discov 2019;9(6):e1329. http://dx.doi.org/10.1002/widm.1329, arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1329, https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1329.

[6] Kaushik M, Sharma R, Fister, Jr. I, Draheim D. Numerical association rule mining: a systematic literature review. 2023, arXiv preprint arXiv:2307.00662.

[7] Rudin C. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nat Mach Intell 2019;1(5):206–15.

[8] Kwon Y-D, Choo J, Yoon I, Park M, Park D, Gwon Y. Matrix encoding networks for neural combinatorial optimization. Adv Neural Inf Process Syst 2021;34:5138–49.

[9] Schuetz MJ, Brubaker JK, Katzgraber HG. Combinatorial optimization with physics-inspired graph neural networks. Nat Mach Intell 2022;4(4):367–77.

[10] Karabulut E, Groth P, Degeler V. 3k: Knowledge-enriched digital twin framework. In: Proceedings of the 14th international conference on the internet of things. 2024, p. 188–93.

[11] Karabulut E, Groth P, Degeler V. Neurosymbolic association rule mining from tabular data. In: Proceedings of the 19th international conference on neurosymbolic learning and reasoning (neSy). 2025, [in press], arXiv:2504.19354.

[12] Bank D, Koenigstein N, Giryes R. Autoencoders. In: Machine learning for data science handbook: data mining and knowledge discovery handbook. 2023, p. 353–74.

[13] Srikant R, Vu Q, Agrawal R. Mining association rules with item constraints. In: Kdd, vol. 97, 1997, p. 67–73.

[14] Baralis E, Cagliero L, Cerquitelli T, Garza P. Generalized association rule mining with constraints. Inform Sci 2012;194:68–84.

[15] Shabtay L, Fournier-Viger P, Yaari R, Dattner I. A guided fp-growth algorithm for mining multitude-targeted item-sets and class association rules in imbalanced data. Inform Sci 2021;553:353–75.

[16] Fournier-Viger P, Wu C-W, Tseng VS. Mining top-k association rules. In: Advances in artificial intelligence: 25th Canadian conference on artificial intelligence, Canadian AI 2012, toronto, on, Canada, May (2012) 28-30. proceedings 25. Springer; 2012, p. 61–73.

[17] Zaki MJ, Hsiao C-J. Charm: An efficient algorithm for closed itemset mining. In: Proceedings of the 2002 SIAM international conference on data mining. SIAM; 2002, p. 457–73.

[18] Yan X, Zhang C, Zhang S. Confidence metrics for association rule mining. Appl Artif Intell 2009;23(8):713–37.

[19] Vincent P, Larochelle H, Bengio Y, Manzagol P-A. Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th international conference on machine learning. 2008, p. 1096–103.

[20] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: An imperative style, high-performance deep learning library. In: Advances in neural information processing systems, vol. 32, Curran Associates, Inc.; 2019, p. 8026–37.

[21] pandas development team T. pandas-dev/pandas: Pandas. 2020, http://dx.doi.org/10.5281/zenodo.3509134.

[22] Kelly M, Longjohn R, Nottingham K. The uci machine learning repository. 2023, https://archive.ics.uci.edu.

[23] Foorthuis R. The impact of discretization method on the detection of six types of anomalies in datasets. In: Proceedings of the 30th benelux conference on artificial intelligence. 2018.

[24] Angelino E, Larus-Stone N, Alabi D, Seltzer M, Rudin C. Learning certifiably optimal rule lists for categorical data. J Mach Learn Res 2018;18(234):1–78.

[25] Liu B, Hsu W, Ma Y. Integrating classification and association rule mining. In: Proceedings of the fourth international conference on knowledge discovery and data mining, KDD. 1998, p. 80–6.

[26] Stupan Ž, Fister I. Niaarm: a minimalistic framework for numerical association rule mining. J Open Source Softw 2022;7(77):4448.

[27] Fister, Jr. I, Fister D, Iglesias A, Galvez A, Osaba E, Del Ser J, et al. Visualization of numerical association rules by hill slopes. In: International conference on intelligent data engineering and automated learning. Springer; 2020, p. 101–11.

[28] Raschka S. Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. J Open Source Softw 2018;3(24). http://dx.doi.org/10.21105/joss.00638.

[29] Fournier-Viger P, Gomariz A, Gueniche T, Soltani A, Wu C-W, Tseng VS, et al. Spmf: a java open-source pattern mining library. J Mach Learn Res 2014;15(1):3389–93.

[30] Dias JR. pyECLAT: association analysis using the eclat method in python. 2020, python package version 1.0.2, BSD 2-Clause License. https://github.com/jeffrichardchemistry/pyECLAT. [Accessed on 15 August 2025].

[31] Hahsler M. Arulespy: exploring association rules and frequent itemsets in python. 2023, arXiv preprint arXiv:2305.15263.

[32] Borgelt C. An implementation of the fp-growth algorithm. In: Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations. 2005, p. 1–5.

[33] Berteloot T, Khoury R, Durand A. Association rules mining with auto-encoders. In: International conference on intelligent data engineering and automated learning. Springer; 2024, p. 51–62.

[34] Hahsler M, Chelluboina S, Hornik K, Buchta C. The arules r-package ecosystem: analyzing interesting patterns from large transaction data sets. J Mach Learn Res 2011;12:2021–5.

[35] Vrbančič G, Brezočnik L, Mlakar U, Fister D, Fister, Jr. I. NiaPy: Python microframework for building nature-inspired algorithms. J Open Source Softw 2018;3. http://dx.doi.org/10.21105/joss.00613.

[36] Mlakar U, Fister, Jr. I, Fister I. Niaautoarm: Automated framework for constructing and evaluating association rule mining pipelines. Mathematics 2025;13(12):1957.

[37] Fister, Jr. I, Salcedo-Sanz S, Alexandre-Cortizo E, Novak D, Fister I, Podgorelec V, et al. Toward explainable time-series numerical association rule mining: A case study in smart-agriculture. Mathematics 2025;13(13):2122.

[38] Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. ACM Sigmod Rec 2000;29(2):1–12.

[39] Zaki MJ, Parthasarathy S, Ogihara M, Li W, et al. New algorithms for fast discovery of association rules. In: KDD, vol. 97, 1997, p. 283–6.

[40] Pei J, Han J, Lu H, Nishio S, Tang S, Yang D. H-mine: Hyper-structure mining of frequent patterns in large databases. In: Proceedings 2001 IEEE international conference on data mining. IEEE; 2001, p. 441–8.

[41] Storn R, Price K. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. J Global Optim 1997;11(4):341–59.

[42] Kennedy J, Eberhart R. Particle swarm optimization. In: Proceedings of iCNN'95-international conference on neural networks, vol. 4, ieee; 1995, p. 1942–8.

[43] Fister I, Iglesias A, Galvez A, Del Ser J, Osaba E, Fister I. Differential evolution for association rule mining using categorical and numerical attributes. In: Intelligent data engineering and automated learning–IDEAL 2018: 19th international conference, madrid, Spain, November (2018) 21–23, proceedings, part i 19. Springer; 2018, p. 79–88.

[44] Gao H, Korn JM, Ferretti S, Monahan JE, Wang Y, Singh M, et al. High-throughput screening using patient-derived tumor xenografts to predict clinical trial drug response. Nature Med 2015;21(11):1318–25.

[45] Ruiz C, Ren H, Huang K, Leskovec J. High dimensional, tabular deep learning with an auxiliary knowledge graph. Adv Neural Inf Process Syst 2023;36:26348–71.

[46] Borgelt C. Pyfim: frequent item set mining algorithms. 2020, http://www.borgelt.net/pyfim.html. Accessed 11 July 2025.

[47] Fournier-Viger P, Gomariz A, Gueniche T, Soltani A, Wu C-W, Tseng VS, et al. Spmf: a java open-source pattern mining library. J Mach Learn Res 2014;15(1):3389–93.