



Human-centric algebraic machine learning

AML-DL Interpreter - Initial

Deliverable D3.5

WP3 - AML Description Language



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952091.

Consortium



ALGEBRAIC AI



Chantalimaud
Foundation



German
Research Center
for Artificial
Intelligence



TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN



Project

Title: ALMA: Human Centric Algebraic Machine Learning
 Acronym: ALMA
 Coordinator: Proyectos y Sistemas de Mantenimiento SL (eProsima)
 Grant agreement: 952091
 Call: H2020-EIC-FETPROACT-2019
 Program: Horizon 2020
 Start: 01 September 2020
 Duration: 48 months
 Website: <https://alma-ai.eu>
 E-mail: alma@eprosima.com
 Consortium: **Proyectos y Sistemas de Mantenimiento SL (eProsima) (EPROS)**, Spain
Champalimaud Foundation (CF), Portugal
ALGEBRAIC AI (ALG), Spain
German Research Center for Artificial Intelligence (DFKI), Germany
Inria (Inria), France
TU Kaiserslautern (TUK), Germany
FIWARE Foundation e.V. (FF), Germany
Technical Research Centre of Finland (VTT), Finland
Carlos III University (UC3M), Spain

Deliverable

Number: **D3.5**
 Title: **AML-DL Interpreter - Initial**
 Month: 18
 Work Package: WP3 - AML Description Language
 Work Package leader: Algebraic AI
 Deliverable leader: Algebraic AI
 Deliverable type: DEM
 Dissemination level: CO
 Date of submission: 2022-02-28
 Version: v1.0
 Status: Finished

Version history

Version	Date	Responsible	Author/Reviewer	Comments
v0.1	2021-07	ALG	Nabil Abderrahaman	Initial draft
v0.2	2022-01	ALG	Nabil Abderrahaman	Review
v0.3	2022-02	ALG	Nabil Abderrahaman	Review
v0.4	2022-02	ALG	Fernando Martin-Maroto	Review

Executive summary

Algebraic Machine Learning (AML) is a relatively new machine learning technique based on algebraic representations of data. Unlike statistical learning, AML algorithms are robust regarding the statistical properties of the data and are parameter-free. The aim of the EU-funded ALMA project is to leverage AML properties to develop a new generation of interactive, human-centric machine learning systems. These systems are expected to reduce bias and prevent discrimination, remember what they know when they are taught something new, facilitate trust and reliability and integrate complex ethical constraints into human-artificial intelligence systems. Furthermore, they are expected to promote distributed, collaborative learning.

This report corresponds to *Deliverable D3.5 - AML-DL Interpreter - Initial* of the ALMA project. This deliverable describes the architecture and usage of the AML-DL Interpreter (AML-DL Interpreter) that can transform human-readable semantic embeddings, written in AML Description Language (AML-DL), into AML input data for the AML Engine. The commands that the interpreter provides are described, and examples of use are provided for each individual one. The last section proposes some possible enhancements for future releases. Together with this document, the Python module that implements the interpreter is provided, as well as example embeddings.

Contents

Executive Summary	3
Contents	4
Acronyms	5
1 Introduction	6
2 AML-DL Interpreter	6
2.1 AML-DL Interpreter architecture overview	6
2.2 Command Set	6
2.2.1 Embedding creation	7
2.2.2 Constant creation	7
2.2.3 Duple creation	7
2.2.4 Term creation and manipulation	8
2.2.5 Auxiliary commands	9
2.2.6 Loading an embedding	9
3 Demonstrator	10
3.1 Instructions	10
3.2 Examples	10
4 Future work	10



Acronyms

AML	Algebraic Machine Learning.
AML-DL	AML Description Language.
AML-DL Interpreter	AML-DL Interpreter.

1 Introduction

The primary goals of AML-DL are to give the user the capability to concisely describe elaborate algebraic semantic embeddings, and to establish a methodology to build embeddings for real life problems. The specification of AML-DL is described in *Deliverable D3.1 - AML-DL Specification - Initial*.

The AML-DL Interpreter implements the AML-DL specification into an interpreter that can transform human-readable semantic embeddings into AML input data for the AML Engine.

2 AML-DL Interpreter

The AML-DL Interpreter is provided under the form of a Python module and strictly follows the AML-DL specification as described in *Deliverable D3.1 - AML-DL Specification - Initial* and *Deliverable D3.2 - AML-DL Specification - Revised*. Using a base language such as Python has obvious advantages as the entire base language is available as a platform to add control flow, variable handling, etc. Relying on a base language also makes the implementation simpler as it makes unnecessary to use a parser or a lexer. Python has the additional advantage of being well-maintained, easy to use, and widely used in data science and machine learning.

The implementation of AML-DL, as described in the specification, is highly customizable, but also instructions perform relatively simple operations. This can lead to the repetition of certain instructions and the appearance of common patterns. The use of a base language can help with code reusability as these repeated instructions and patterns can be grouped into functions and classes. In the future, if certain patterns are common to a wide range of problem, they can be added to the AML-DL Python module.

2.1 AML-DL Interpreter architecture overview

The output of the AML-DL Interpreter is comprised of a set of descriptors representing constants, and a set of duples, which are formed by a pair of sets of descriptors. Descriptors can contain constants or other nested descriptors, giving rise to a tree structure. As the leaves of this tree structure is always formed by constants, duples can in essence be assimilated to a pair of sets of constants.

Commands are either final or bound. Commands are *final* if their return descriptors are directly appended to the embedding's output. These commands are **C**, **CV**, **ADD**. Commands are *bound* if their return descriptors are not appended to the embedding's output. These descriptors are meant to be used as inputs for other commands and bound to the output at a later step.

At any point, the program state corresponds to a set of final descriptors collected so far, and an internal dictionary that maps names to descriptor objects. Names are regular character string that can be used to invoke descriptor objects by name.

2.2 Command Set

The functions and classes of the AML-DL Interpreter are defined in the module `aml.amlidl`. They can be individually imported with

```
1 from aml.amlidl import (
2     load_embedding,
3     Descriptor,
4     ADD, APP, C, CMP, CV, E, EXC, F, HEADER,
5     INC, M, R, S, SOME, T, V
6 )
```

In this document, it is assumed that they are all imported into the main namespace with

```
1 from aml.aml1 import *
```

2.2.1 Embedding creation

Most function need to be called in a Python Context that represents an embedding. An embedding is an instance of the class Descriptor.

```
1 with Descriptor() as theEmbedding:
2     ...
```

All following examples are assumed to be executed within such a Python Context.

All duples are assigned the region of their embedding at the time they are created. The REGION property of the embedding can be modified at any point.

```
1 with Descriptor() as theEmbedding:
2     theEmbedding.REGION = 5
```

2.2.2 Constant creation

C(name)

Defines a constant with the given name.

```
1 C("Q")
```

CV(name, n)

Defines a vector of constants with names: name[0], name[1], ..., name[n-1].

```
1 CV("Q", 10)
```

V([name])

Defines an empty vector.

```
1 V("Qx")
2 APP(F("Qx"), V())
```

2.2.3 Duple creation

INC([name])

Defines a positive duple $l \leq h$.

l and h cannot be duples.

```
1 ADD(INC("U", B))
```

EXC([name])

Defines a positive duple $l \leq h$.

l and h cannot be duples.

```
1 ADD(EXC("Qx", "Qy"))
```

2.2.4 Term creation and manipulation

APP(p, a)

Appends a to p, where \verbp— is of type vector.

The vector p is modified in place.

```

1 # Create a vector that contains two empty vectors
2 V("Qx")
3 APP(F("Qx"), V())
4 APP(F("Qx"), V())

```

M(p₁, ..., p_n)

Returns a set containing the p₁ to p_n elements.

If some arguments is of type *tvector*, returns a descriptor of type *tvector* whose components are sets.

```

1 M(Qx, Qy, B, U)

```

R(p, r)

Removes r from the vector p.

If r is an integer, removes the *r*-th element of p.

Returns a new vector, and p is not modified.

```

1 b = CV("black", 2)
2 w = CV("white", 2)
3 wb[0] = M(w, R(b, 0)) # w[0], w[1], b[1]
4 wb[1] = M(w, R(b, 1)) # w[0], w[1], b[0]

```

CMP(p₁, p₂)

Declares p₁ and p₂ as the logical negations of each other.

Operates component-wise if p₁ and p₂ are vectors of the same length.

No return value.

```

1 CV("Q", 5)
2 CV("E", 5)
3 CMP("Q", "E")

```

CMP(p)

Finds or calculates the complementary of p.

Returns a copy of p where every element is substituted by its complementary.

```

1 CV("Q", 5)
2 CV("E", 5)
3 CMP("Q", "E")
4 F("E") == CMP("Q")

```

T(p [,i])

Associates an iterator to p, where p cannot be a duple.

The optional parameter i is an integer that identifies the iterator.

Returns a *tvector*.

```

1  b = CV("black", 2)
2  w = CV("white", 2)
3  wb = M(w , R(b, T(b)))
4  # wb.t[0] = M(w , R(b, 0)) = w[0], w[1], b[1]
5  # wb.t[1] = M(w , R(b, 1)) = w[0], w[1], b[0]

```

2.2.5 Auxiliary commands

F(name [,i])

Returns the descriptor associated with name.

If i is given and p is a vector, return the i-th element of p.

```

1  CV("black", 5)
2  b3 = F("black", 3)

```

SOME(p, prob [,atLeastOne, notAll])

Randomly chooses items from p with a probability **prob** per element.

When set to True, the flags ensure that there is at least one element, and not all have been chosen, respectively. By default they are set to False.

```

1  CV("black", 5)
2  SOME("black", 0.5)

```

HEADER(name)

Guards a section of the embedding that should be read just once.

Returns True the first time, False otherwise.

```

1  if HEADER("black constants"):
2      CV("black", 5)

```

2.2.6 Loading an embedding

The embedding can be wrapped in a function **embedding** that takes any number of input arguments, that can be used as parameters, and returns the embedding's output.

```

1  def embedding():
2      with Descriptor() as theEmbedding:
3          C("A")
4          C("B")
5          ADD(INC("A", "B"))
6
7          constantNames = [el.key for el in F(theEmbedding.PEND_TRANSF).r]
8          p = F(theEmbedding.INCLUSIONS).r
9          n = F(theEmbedding.EXCLUSIONS).r
10
11     return (constantNames, p, n)

```

From a different script the embedding can be loaded using the **load_embedding** function.

```

1  import aml.amlddl as dl
2  (constantNames, p, n) = dl.load_embedding("path/to/embedding.py")

```



3 Demonstrator

3.1 Instructions

Instructions for installation and usage are included in `README.md` in the project folder.

The Python library is provided as a binary file, and can be directly executed in most Linux distributions. However, to avoid compatibility issues, the recommended approach is to use a Docker container. Instructions for running the AML-DL Interpreter in a Docker container are also provided in `README.md`.

3.2 Examples

The interpreter is distributed with two full semantic embeddings written in AML-DL, and an example script, `interpreter.py` that simplifies the task of reading embeddings.

These examples provide an overview of how to use the library to build a semantic embedding and how to load the resulting AML input data.

The script `interpreter.py` can be used from the command line. Full documentation can be found invoking the `--help` flag. The two examples provided can be run with the following commands (where `N` represents the size of the chessboard), respectively:

```
1 $ python interpreter.py -o OutputFile.out -i embeddings/queensEmbedding.py N
2
3 $ python interpreter.py -o OutputFile.out -i embeddings/cyclicGroupEmbedding.py
```

4 Future work

The implementation of AML-DL presented in this work is highly customizable. It provides the user with a powerful tool to describe embeddings. As common patterns appear, they can be identified and added to the tool, speeding up the process and reducing the amount of boilerplate code in the semantic embeddings. Regarding the current Python implementation, where the interpreter is under the form of a module, common patterns can be directly added to it as submodules, or distributed as independent modules.