MLearn: A Declarative Machine Learning Language for Database Systems

Maximilian E. Schüle
m.schuele@tum.de
Matthias Bungeroth
matthias.bungeroth@tum.de
Alfons Kemper
alfons.kemper@tum.de
Stephan Günnemann
stephan.guennemann@tum.de
Thomas Neumann
thomas.neumann@tum.de
Technical University of Munich

ABSTRACT
This paper outlines the requirements of our ML2SQL compiler that allows a dedicated machine learning language (MLearn) to be run on different target architectures. The language was designed to cover an end-to-end machine learning process, including initial data curation, with the focus on moving computations inside the core of database systems. To move computations to the data, we explain the architecture of a compiler that translates into target specific user-defined-functions for the PostgreSQL and HyPer database systems. For computations inside user-defined-functions, we explain the necessary tensor datatypes and the corresponding functions. We base the explanations on an accompanying example of linear regression. To face the challenges to database systems arising from array-like data, we propose such solutions as integrating ArrayQL as stored procedures to unify the relational and array perspectives.

CCS CONCEPTS
• Information systems → Main memory engines.

KEYWORDS
SQL, Declarative Language, Database Scripting Languages

1 INTRODUCTION
Data for machine learning has to be preprocessed until it is possible to apply linear algebra to features aggregated to matrices. For data curation, text-like data such as CSV files have to be processed and cut until a suitable format is found. Afterwards, such optimisation methods as gradient descent find the optimal weights for a parameterised loss function in order to allow predictions on unlabelled data. From a systems developer point of view, database systems form the native way of efficiently storing data in index structures. Inside of database systems, SQL, as the declarative language, simplifies data curation because it allows feature extraction as projections and selections of the only relevant tuples by design. In addition, hybrid main-memory database systems such as HyPer [4] combine transactional and analytical workload to avoid separate systems for each domain. However, for machine learning purposes, data still has to be extracted before.

In the last decade, many studies have presented architectures for building end-to-end machine learning systems [1, 3, 9]. To assist computations in database systems, the support of matrix or tensor algebra is essential. Therefore, different systems tackle the challenges of representing arrays natively...
in database systems. Array database systems such as RasDaMan [2] or SciDB [8] replace tables by arrays as the native way of storing data.

As we want to move computations to the data, we have analysed the power of scripting languages within database systems. We argued that tensor operations form the main building for database systems, which is why we have demonstrated the ML2SQL compiler [7] that translates a dedicated machine learning language (MLean) into domain-specific database language extensions. This paper focuses on the architectural stack for compiling the language (s. Figure 1) and executing the result as stored procedures inside of database systems.

The paper’s main contributions are the description of the architecture behind MLean with an accompanying example, an extension of PostgreSQL by linear algebra and gradient descent on array datatypes, as well as a look on integrating array datatypes.

This work is structured as follows: firstly, the MLean language is introduced together with the functionality of the ML2SQL compiler for translation into Python, pl/pgSQL, or HyPerScript. Therefore, we will provide an example of linear regression using data read from a CSV file to show the modularity of the language. Then we will show how the ML2SQL compiler translates these statements into the target language, here as user-defined functions for PostgreSQL using pl/pgSQL. We therefore explain the technical background of features we have added, in order to run different workloads.

2 THE MLEAN LANGUAGE

![Figure 2: Compiling the MLean language with the ML2SQL compiler (dark blue): it first preprocesses import and include statements, then it compiles to SQL or Python code.](image)

The dedicated machine learning language MLean is aimed at preprocessing data and training models intuitively by minimising the overhead. It provides building blocks for data loading as well as for k-fold cross validation. The underlying models can be optimised by gradient descent or numerically using matrix operations. In addition, the language allows the definition of customised building blocks as functions based on procedural statements and tensor algebra. The classical approach of working on data is to use Python with libraries such as Pandas (for data analysis), NumPy (for matrix operations) and TensorFlow (for optimisations). We simplify the use of these by providing building blocks that make use of the libraries stated when using Python. For example, NumPy computations and TensorFlow models for gradient descent can be specified in linear algebra instead of creating an expression tree manually.

Listing 1: CSV file loading in MLean: the building block requires the target name, the source file, the necessary columns and the delimiter; it allows preprocessing options.

Listing 1 shows the building block for loading CSV files. It requires the target name (to create tensors with NumPy or relations in database systems), the name of the source file, the necessary columns and the delimiter, as well as the preprocessing options such as the deletion of certain entries or string replacements. When compiling for SQL, the resulting code creates a new table, copies CSV files with domain-specific commands while the preprocessing options are translated in update or delete queries (s. Listing 2).

Listing 2: The compiled code for loading CSV files in PostgreSQL: create, update and delete SQL statements are used, as well as the copy command.

Actually, MLean is designed as a meta-language to facilitate the use of database scripting languages or Python
frameworks. The ML2SQL compiler\(^1\) (s. Figure 2) produces Python or SQL code runnable in PostgreSQL or HyPer. The compiler is written in ANTLR [5] and allows import and include statements. As the other computations rely on the initial data loading, the modular language design will allow the inclusion of C preprocessor statements (internally, it uses the gcc \(-E\) command). We allow two kinds of import statements, include and import. The first command allows basic text insertions known from the gcc preprocessor to load self-defined modules. The second command imports predefined libraries for time measurements, distributions and other convenient functions. In Listing 3, we make use of the import statements to perform linear regression numerically: we first load the libraries for time measurements, then we include the predefined building blocks for loading a training and a test dataset. Afterwards, we define tensors for the attributes (\(X\)) and the labels (\(y\)); then we add one dimension to \(X\) for the bias and finally we compute the optimal weights, all using tensor algebra.

```
import time
import regression
import functions
#include "include/loadTaxiData.ml"
#include "include/loadTestTaxiData.ml"
create tensor DATA from taxiData(trip_seconds, trip_miles, pickup_community_area, dropoff_community_area, fare, tips)
create tensor TEST_DATA from taxiDataTest(trip_seconds, trip_miles, pickup_community_area, dropoff_community_area, fare, tips)
X = DATA[:, 0:4]
y = DATA[:, 5]
X_test = TEST_DATA[:, 0:4]
y_test = TEST_DATA[:, 5:5]
start_bias = time()
X = addBiasTerm(X)
start_reg = time()
w = regression(X, y)
start_pred = time()
err = predict(X_test, w) - y_test
end_reg = time()
end_pred = time()
end_bias = time()

Listing 3: Numerical linear regression in MLearn.
```

### 3 TECHNICAL BACKGROUND

In order to allow machine-learning-related computations within database systems, they have to provide tensors and functionalities for training a model. HyPer has already extended its array datatype to serve as tensors by allowing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar mul.</td>
<td>(a \times b)</td>
<td>(a \in \mathbb{R}, b \in \mathbb{R}^{n \times m})</td>
<td>(\mathbb{R}^{n \times m})</td>
</tr>
<tr>
<td>tensor mul.</td>
<td>(a \times b)</td>
<td>(a \in \mathbb{R}^{n \times k}, b \in \mathbb{R}^{o \times m})</td>
<td>(\mathbb{R}^{o \times k})</td>
</tr>
<tr>
<td>tensor add.</td>
<td>(a + b)</td>
<td>(a, b \in \mathbb{R}^{n \times m})</td>
<td>(\mathbb{R}^{n \times m})</td>
</tr>
<tr>
<td>tensor sub.</td>
<td>(a - b)</td>
<td>(a, b \in \mathbb{R}^{n \times m})</td>
<td>(\mathbb{R}^{n \times m})</td>
</tr>
<tr>
<td>tensor power</td>
<td>(a^b)</td>
<td>(a \in \mathbb{R}^{n \times m}, b \in \mathbb{Z})</td>
<td>(\mathbb{R}^{o \times m})</td>
</tr>
<tr>
<td>transpose</td>
<td>(a^T)</td>
<td>(a \in \mathbb{R}^{n \times m})</td>
<td>(\mathbb{R}^{m \times n})</td>
</tr>
<tr>
<td>identity</td>
<td>(id(n))</td>
<td>(n \in \mathbb{N})</td>
<td>(\mathbb{R}^{n \times n})</td>
</tr>
<tr>
<td>array fill</td>
<td>(fill(r, n, m))</td>
<td>(r \in \mathbb{R}, n, m \in \mathbb{N})</td>
<td>(\mathbb{R}^{n \times m})</td>
</tr>
</tbody>
</table>

Table 1: Implemented matrix operations for PostgreSQL needed by MLearn.

algebra on those types. Inside select clauses of query statements, linear algebra on array datatypes allows various computations. To reach a broader audience for our declarative machine learning language, we also provide some matrix algebra functionalities for PostgreSQL online\(^2\). The operations are first precompiled as a shared library (s. Listing 4), then loaded by SQL statements (s. Listing 5). Table 1 shows the list of matrix operations needed that we implemented in PostgreSQL to allow pl/pgSQL procedures with matrix operation produced by the ML2SQL compiler.

```
Datum tensor_add(PG_FUNCTION_ARGS)
{
...
ArrayType *a1 = PG_GETARG_ARRAYTYPE_P(0),
.FLOAT
.FLOAT

Listing 4: Exemplary tensor operation (add) programmed in C for PostgreSQL.
```

```
CREATE or REPLACE FUNCTION tensor_add(FLOAT [], FLOAT [])
RETURNS FLOAT [] AS
'./tmp/psql-matrix-extension/bin/MatrixOperations.so',
'tensor_add' LANGUAGE C STRICT;
```

Listing 5: Operator creation for addition on arrays in PostgreSQL out of the shared library.

In addition to matrix operations, a gradient descent optimizer is essential for training models inside database systems. When compiling MLearn to SQL, we make use of our gradient descent operator supporting automatic differentiation

\(^1\)https://gitlab.db.in.tum.de/ml2sql/ml2sql

\(^2\)https://gitlab.db.in.tum.de/ml2sql/psql-matrix-extension
with lambda expressions already integrated in HyPer [6]. Lambda expressions inject user-defined code into hard-coded database operators, for example distance metrics in clustering algorithms. Here, they allow arbitrary loss functions to be specified as shown in Listing 6. Inside, we can combine attributes from a training dataset with weights, either individually by hand or aggregated to matrices.

Listing 6: Gradient descent table function using lambda expressions: inside the expression, each parameter can be combined separately or using matrix algebra.

In addition to matrix operations, we adapted our gradient descent framework to run with PostgreSQL. However, as the dimensions of PostgreSQL arrays are assigned dynamically, the PostgreSQL gradient descent operator expects the dimensions as futher arguments inside the lambda expression for the loss function. Together with these extensions, we are now able to translate the code in Listing 3 to pl/pgSQL to run in PostgreSQL. Listing 7 shows an extract of the resulting code.

Listing 7: Resulting pl/pgSQL procedures for linear regression and increasing the matrix by the bias.

4 CONCLUSION AND ONGOING WORK

This paper has introduced the architectural details behind a declarative machine learning language (MLearn) with the aim of shifting computations inside of the core of database systems. It has shown how the ML2SQL compiler treats preprocessor statements to allow the inclusion of code snippets and libraries. It has provided a basic example of how to compute linear regression and referred to the necessary technical implementations such as gradient descent in PostgreSQL and HyPer. For this purpose, we have specified the required matrix operations in PostgreSQL with one coding example.

To sum up, we have discovered out that array processing represents the major building block for tasks related to machine learning. These tasks would strongly benefit from SQL especially for data preprocessing. In addition, when integrating the advantages of array database into hybrid OLTP and OLAP database systems, no domain specific systems would be required. We shall therefore work on regarding whole tables as matrices in stored procedures that are written in ArrayQL.