



A survey on machine learning in array databases

Sebastián Villarroya^{1,2} · Peter Baumann³

Accepted: 7 July 2022
© The Author(s) 2022

Abstract

This paper provides an in-depth survey on the integration of machine learning and array databases. First, machine learning support in modern database management systems is introduced. From straightforward implementations of linear algebra operations in SQL to machine learning capabilities of specialized database managers designed to process specific types of data, a number of different approaches are overviewed. Then, the paper covers the database features already implemented in current machine learning systems. Features such as rewriting, compression, and caching allow users to implement more efficient machine learning applications. The underlying linear algebra computations in some of the most used machine learning algorithms are studied in order to determine which linear algebra operations should be efficiently implemented by array databases. An exhaustive overview of array data and relevant array database managers is also provided. Those database features that have been proven of special importance for efficient execution of machine learning algorithms are analyzed in detail for each relevant array database management system. Finally, current state of array databases capabilities for machine learning implementation is shown through two example implementations in Rasdaman and SciDB.

Keywords Array data · Array database managers · Machine learning · Efficient array machine learning

1 Introduction

Machine learning (ML) has become in recent years an essential tool not only in the field of academic research but also in business and industrial applications. Due to the improvement experienced by hardware devices in recent years, an increasing number of companies in many different business areas have decided to take advantage of ML technologies to analyze the large amounts of data they need to manage.

For many years a huge research effort has been devoted to the integration of ML tools and relational database management systems (RDBMSs), either implementing ML algorithms in RDBMSs [1–9] or implementing well known database optimization techniques in ML tools [10–16].

However, this research effort devoted to relational databases has not been done in the field of array databases until very recently [17–19]. Taking into account that increasingly large amounts of data are generated every day and a relevant part of them are array data, it is essential to leverage the efficiency of array databases for the analysis of array data in ML applications. Therefore, the most efficient tools in array data analysis will allow users to boost analysis capabilities in many and diverse application domains such as cancer detection, pollution analysis, weather forecasting and environmental classification.

This paper aims to provide a broad overview of the current state of the integration of ML algorithms and array databases.

First, an overview of recent solutions and approaches providing support for ML algorithms in RDBMSs is introduced. We begin with straightforward SQL implementations of basic linear algebra operations, e.g., matrix multiplication. Then, more efficient solutions make use of *user-defined functions* (UDFs). Beyond UDFs, *user-defined*

Peter Baumann contributed equally to this work.

✉ Sebastián Villarroya
s.villarroya@usc.es

Peter Baumann
p.baumann@jacobs-university.de

¹ Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, 15782, Spain

² Work performed while at Jacobs University Bremen, Bremen, Germany

³ Department of Computer Science and Electrical Engineering, Jacobs University Bremen, Campus Ring 1, Bremen, 28759, Germany

aggregates (UDAs) have shown to be very useful to implement those ML algorithms that need to iterate over large datasets, e.g., *Gradient Descent* and *k-means*. More complex solutions are focused on leveraging the underlying data model of RDBMSs when implementing ML algorithms, e.g., *learning over joins* and *statistical relational learning* (SRL). Finally, those solutions that provide deeper modifications to RDBMSs are introduced. Here, we have specialized systems for the analysis of specific data types, e.g., array DBMSs.

Then, an overview of main database features implemented in relevant ML systems is provided. Classical logical rewrites (e.g. loop vectorization and algebraic simplification) have shown to be relevant in ML systems. Also physical rewrites (e.g., flow rewrites, execution type and physical operation selection) are of key importance in ML implementations. Similarly to data access methods in databases, ML tools can leverage data access optimization techniques (e.g. compression and caching) to implement more efficient ML algorithms.

The underlying linear algebra of three widely used ML algorithms, namely *linear regression*, *logistic regression* and *feed-forward neural networks*, has been extensively studied to determine the linear algebra operations. Efficient implementations of such operations are required in array databases in order to provide a efficient performance of ML algorithms on array data.

A brief introduction on array data and array databases is also provided. The array data model and main database properties (e.g., parallelization and query rewriting) are specifically pointed out for array databases. Next, several state-of-art approaches for array data processing are classified into groups depending on their characteristics. A description of main features is also provided for each of them. Then, the aforementioned database features that are relevant for ML algorithms are used to provide a detailed comparison between such different solutions. Finally, examples of ML algorithm implementation are discussed for two relevant array databases.

2 Motivating examples

Many application domains require from machine learning analytics. And many of such application domains need to process very large amounts of array data.

For example, in geoscience applications we can find challenging problems that naturally fit many machine learning methods.

- **Estimating variables from observations.** Climate events such as tornadogenesis and cyclogenesis can be

detected by analyzing array datasets. Also, they can be predicted by discovering their precursors. Examples are [20, 21].

- **Characterizing objects and events.** Some critical geoscience variables (e.g., methane concentrations in air) are difficult to be monitored directly. Thus, machine learning methods can be used to infer such variables by analyzing other variables collected via satellite or ground sensors. An example approach is [22].
- **Long-term forecasting.** Early adaptation policies and resource planning can be devised if prediction of geoscience variables are provided ahead of time. Machine learning methods for forecasting climate variables using the spatial and temporal structure of geoscience data have been explored in recent works such as [23].

Life sciences also leverage analytics on array data. For example, machine learning systems have multiple applications in medical imaging for different scenarios.

- **Screening.** ML is used as a screening tool to classify studies according to the probability of the presence of a disease. Application examples are: breast cancer screening with mammography [24], and early diagnosis of Alzheimer's disease [25].
- **Replacement.** There are some areas in which ML has the ability to replace doctors. Examples are: estimation of bone age by artificial intelligence software [26], calculation of breast radiological density and risk scoring [27].
- **Complementation.** In this scenario there is an analysis and a report by the doctor, and a complementary contribution of the machine learning system. Examples are: registration and segmentation, interpretation and automatic classification, radiation dose calculation, automatic data integration, improvement in the quality of radiological reports.

3 ML support in DBMS

Traditionally, database management systems have been used in ML as simple data stores. ML algorithms pull data out from databases, and pre-process and analyze them. However, as stated in [28], performing ML inside a database system has some advantages. First, applying data processing capabilities of DBMSs (e.g., storage, management, transformation and querying) to operations over ML models (e.g., selecting, training, deploying, updating and managing) is a *convenient* way to obtain an end-to-end ML platform. Second, performing data analysis inside the DBMS may improve *efficiency* by (1) avoiding to copy

massive amounts of data to external systems, (2) leveraging their efficient and scalable data processing techniques, and (3) allowing optimization across tasks. Finally, *declarative* task specification might be adequate for ML, enabling automatic optimization and simplifying development.

3.1 SQL implementation

Matrices and linear algebra are on the basis of many ML algorithms. As a first straightforward approach to the implementation of ML in RDBMSs, tables can be used to store matrices, and linear algebra operations can be expressed in SQL. This solution is considered in systems like [4] and [2]. An example of this basic approach is the matrix multiplication with sparse representation. Considering matrices $A_{m \times l}$ and $B_{l \times n}$, the resulting matrix C is a $m \times n$ matrix where $c_{ij} = \sum_{k=1}^l a_{ik}b_{kj}$. Figure 1(a) shows the tables we may use to represent matrices A and B , where columns i, j store the matrix index, and column val stores the matrix value at this specific index. We can now compute the multiplication with the following SQL query.

```
SELECT A.i, B.j, SUM(A.val * B.val)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i = B.j;
```

Code 1 Matrix multiplication with built-in aggregate function.

i	j	val	i	j	val
1	1	a_{11}	1	1	b_{11}
1	...	...	1	...	...
1	l	a_{1l}	1	n	b_{1n}
...	...	...	...	...	...
m	1	a_{m1}	l	1	b_{l1}
m	...	...	l	...	...
m	l	a_{ml}	l	n	b_{ln}

$$A = \begin{pmatrix} a_{11} & \dots & a_{1l} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{ml} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{l1} & \dots & b_{ln} \end{pmatrix}$$

(a) Sparse representation

3.2 UDF implementation

Although the previous solution works well for sparse matrices, it is quite inefficient for dense matrices. These inefficiencies can be eliminated using new user data types (UDTs) and functions (UDFs). Systems like [2] and [3] also implement this solution. An example of this approach is the matrix multiplication with vector representation. A new data type *vector* may be defined as a UDT to efficiently store matrix rows and columns. Figure 1(b) shows the tables used to represent matrices A and B . Table A stores the row index i and the relevant row value, *row*. Table B stores the column index j and the relevant column value, *col*. Notice that both *row* and *col* are *vector* UDTs. Defining a UDF *dotproduct*(v_1, v_2) to efficiently compute the dot product of two vectors, we can compute the multiplication with the following SQL sentence:

```
SELECT A.i, B.j, dotproduct(A.row, B.col)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i = B.j;
```

Code 2 Matrix multiplication with UDF function.

3.3 Extensions

Many ML algorithms involve iterative computations over large datasets. Specifically, UDAs have provided an

i	row	i	col
1	$[a_{11}, \dots, a_{1l}]$	1	$[b_{11}, \dots, b_{1n}]$
...	...	...	...
m	$[a_{m1}, \dots, a_{ml}]$	n	$[b_{n1}, \dots, b_{nn}]$

$$A = \begin{pmatrix} a_{11} & \dots & a_{1l} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{ml} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{l1} & \dots & b_{ln} \end{pmatrix}$$

(b) Vector representation

Fig. 1 Matrix storage in tables

impressive ease to implement iterative ML algorithms. For example, [3] and [6] implement the Gradient Descent algorithm following this approach. Another example is the k -means method for clustering implemented in [6] and [7], also following this approach. Iterative algorithms can be implemented using UDAs by specifying the following functions:

- *Init(state)* initializes those variables used to store the state.
- *Accumulate(state, data)* uses the parameter *data* to update the state.
- *Finalize(state)* computes the final result using the state.
- *Merge(state₁, state₂)* merges different state values computed over disjoint input subsets. Only implemented when needed.

In a very recent work [29], a new approach for declarative specification of recursive computations in a RDBMS has been proposed. Multiple versions of a database table, accessed via *array-style* indices, are allowed by extending SQL. For example, in a *feed-forward neural network* [29], dependencies among activations, weights and errors can be declaratively expressed avoiding imperative control flow by writing the forward-backward passes with such SQL extensions. Section 5.3 provides the mathematical foundations of the forward-backward computations in a *feed-forward* neural network.

The forward pass computes the activation values of the neurons at each layer. Table version $A[i][j]$ stores all the activation values of the layer j at the iteration step i . Each $A[i][j]$ is supposed to be “chunked” and stored as matrices ACT in the following table:

$A(ITER, LAYER, ROW, COL, ACT)$

Activations of layer j at learning iteration i are computed as a weighted sum of output values of all the neurons of layer $j-1$ at iteration i . The weighted sums in layer j at the iteration step i are stored in the table $WI[i][j]$. A similar approach is used to store the weighted sums, each “chunk” of $WI[i][j]$ is stored as a matrix VAL in the following table:

$WI(ITER, LAYER, COL, VAL)$

Using the above tables, the forward pass can be specified with the following queries using SQL extensions to allow the recursive computation of activations and weighted sums. The forward pass begins by initializing the first layer of activations with the input data.

```
CREATE TABLE A[i:0...] [j:0] (COL, ACT) AS
SELECT DI.COL, DI.VAL
FROM DATA_INPUT AS DI;
```

Code 3 Query to load the input layer with input data.

Then, the weighted sums for each layer j of the network at iteration i are computed using the weights of layer j and the activations of layer $j-1$ at iteration i .

```
CREATE TABLE WI[i:0...] [j:1...9] (COL, VAL) AS
SELECT W.COL, SUM(matmul(A.ACT, W.MAT))
FROM W[i][j] AS W, A[i][j-1] AS A
WHERE W.ROW = A.COL
GROUP BY W.COL;
```

Code 4 Query to generate the weighted sums of the network.

These weighted sums are then used to compute the subsequent activation values in the network. Thus, activations in layer j at iteration i are computed applying the activation function *relu* to the weighted sums in layer j at iteration i .

```
CREATE TABLE A[i:0...] [j:1...8] (COL, ACT) AS
SELECT WI.COL, relu(WI.VAL + B.VEC)
FROM WI[i][j] AS WI, B[i][j] AS B
WHERE WI.COL = B.COL;
```

Code 5 Query to generate the activations in the hidden layers.

Finally, the activation values of the output layer ($l=9$) at iteration i are computed by applying the function *softmax* to the weighted sums of the output layer at iteration i .

```
CREATE TABLE A[i:0...] [j:9] (COL, ACT) AS
SELECT WI.COL, softmax(WI.VAL + B.VEC)
FROM WI[i][j] AS WI, B[i][j] AS B
WHERE WI.COL = B.COL;
```

Code 6 Query to generate the activations in the output layer.

In the backward pass, the errors computed in the output layer of the network are back-propagated through the network. Similarly to activations and weighted sums, the table version $E[i][j]$ stores the errors in the layer j at learning iteration i . These errors are then used to compute the network weights. Table $W[i][j]$ stores the weights of the model in layer j at iteration i . Both E and W are also “chunked” and stored in the following tables:

$E(ITER, LAYER, COL, ERR)$
 $W(ITER, LAYER, ROW, COL, MAT)$

The backward pass begins with the computation of errors in the output layer ($l=9$). The activation values in the output layer at iteration i and the expected output values are used to compute the errors in the output layer at iteration i .

```
CREATE TABLE E[i:0...] [j:9] (COL, ERR) AS
SELECT A.COL, crossentropyderiv(A.ACT, DO.VAL)
FROM A[i][j] AS A, DATA_OUTPUT AS DO;
```

Code 7 Query to compute the errors in the output layer.

Then, the errors of the subsequent layers are computed. The errors in the layer j at iteration i are computed using the activation values in layer j at iteration i , the weights in layer $j+1$ at iteration i , and the errors in layer $j+1$ at iteration i .

```
CREATE TABLE E[i:0...] [j:1...8] (COL, ERR) AS
SELECT W.ROW, SUM(matmul(E.ERR, t(W.MAT)) * reluderiv(A.ACT))
FROM A[i] [j] AS A, E[i] [j+1] AS E, W[i] [j+1] AS W
WHERE A.COL = W.ROW AND W.COL = E.COL
GROUP BY W.ROW;
```

Code 8 Query to generate the errors in the remainder layers.

Finally, the weights of the model are updated for the current iteration. The weights in layer j at iteration i are computed using the weights in layer j at iteration $i-1$, the errors in layer j at iteration $i-1$ and the activation values in layer $j-1$ at iteration $i-1$.

```
CREATE TABLE W[i:1...] [j:1...9] (ROW, COL, MAT) AS
SELECT W.ROW, W.COL, W.MAT - matmul(t(A.ACT), E.ERR) * 0.00000001
FROM W[i-1] [j] AS W, E[i-1] [j] AS E, A[i-1] [j-1] AS A
WHERE A.COL = W.ROW AND W.COL = E.COL;
```

Code 9 Query to update the weights in the network.

Thus, with extensions in SQL to allow different versions of a database table, accessed via indices i and j , the recursive computations in a *feed-forward* neural network can be specified in a declarative manner.

3.4 Learning “over” joins

Some recent lines of research are focused on leveraging the underlying data model of RDBMS, named *multi-table relational model*, during the implementation of ML algorithms. One of these lines is *learning over joins*, which aims to accelerate ML algorithms by exploiting common database dependencies in multi-table datasets, e.g., key-foreign key dependencies (KFKD) and multi-valued dependencies (MVD). Taking into account that join computation usually introduces redundancy in the data, and in turn redundancy in many ML computations, learning over joins rather than computing the join output will provide faster results. Current systems may be grouped into three categories: (1) specific ML algorithms, e.g., Kumar et al. [30], Schleich et al. [31], Nikolic and Olteanu [32], TensorDB [1], (2) custom libraries for specific ML algorithms, e.g., Rendle [33], Kumar et al. [34], and (3) generalized learning over joins, e.g., Chen et al. [35], Ghoting et al. [36], Li et al. [37], Abo-Khamis et al. [38].

3.5 SRL systems

Another research line which aims to leverage the underlying RDBMS data model is *statistical relational learning* (SRL). The “independently-and-identically distributed”

(IID) assumption, made by standard ML algorithms, cannot be applied to multi-table datasets. One solution to this issue is provided by SRL ML algorithms which do not make the IID assumption and exploit the structure information to learn directly over multi-table datasets. In addition, SRL allows to predict multiple correlated variables simultaneously with one model. Several SRL models have been proposed in the literature, e.g., Markov Logic Network (MLN) [39] and Probabilistic Soft Logic (PSL) [40]. Examples of modern ML systems implementing SRL include [41–44].

3.6 Specialized DBMSs

All the previous systems have been designed for processing structured relational data. However, since ML can be applied to any type of data, many ML applications need to process large amounts of unstructured data. Thus, several ML systems have been built for processing stacks of ML workloads over specific non-relational data types.

3.6.1 Multimedia ML systems

Multimedia query systems have experienced a renewed interest in recent years mainly due to the popularity of deep convolutional neural networks (CNNs) for accurate computer vision. Recent solutions include [45] and [46].

3.6.2 Text ML systems

No native support for ML on text data is provided by most database systems. Thus, some ML systems, such as [47],

have been designed to store, index, and provide analytics over large amounts of text data.

3.6.3 Time series ML systems

Time series datasets have been an important source of information for many years. In recent years they also became important for many ML applications, e.g., weather forecast. Several ML systems aim to optimize the processing of time series data, e.g., [48] and [49].

3.6.4 Graph ML systems

Several graph-oriented ML systems, such as [50], have been proposed in recent years due the increasingly importance of graphs in social media and web applications.

3.6.5 Array ML systems

Many scientific data processing systems have been designed for leveraging the huge amount of large multi-dimensional array datasets generated every day by many monitoring applications. Block-partitioned arrays are managed by their storage layer. Both read and write operations are defined on such array data. Rasdaman [51] is the pioneering array database management system. It supports a SQL-based query language called Raster Query Language (rasQL). Rasdaman also provides APIs for several environments, e.g., R, Python, Java, C++. Simple ML algorithms, such as linear regression, can be defined as *user defined functions* and called directly in rasQL to be executed in Rasdaman. SciDB [52] also supports a SQL-based query language called Array Query Language (AQL) and several APIs. A small number of linear algebra operators allow the execution of simple regression tasks. Section 7 provides an in-depth analysis of the most important database features for ML, introduced in Section 4, currently implemented in modern array database management systems.

4 DB features in ML tools

Database-inspired optimization techniques are applied in many ML systems to improve the efficiency of ML programs. The most important optimization objective is to minimize execution time. Nevertheless, recent optimizers allow for optimization based on different objectives, e.g., minimize time [14], minimize monetary cost [53], maximizing accuracy [54]. To highlight the importance of those database optimization techniques implemented in ML systems, a brief overview of some relevant techniques is provided in this section.

4.1 Logical rewrites

4.1.1 Common subexpression elimination

One of the most used programming language rewrite in ML systems is common subexpression elimination (CSE). In a bottom-up pass through the graph, this optimization technique (1) consolidates leaf nodes, and (2) merges equivalent operations, i.e., those with same inputs and data types. Some operators must be prevented from invalid elimination, e.g., `print`. Thus, operation-specific handling is required for these operations.

4.1.2 Loop vectorization

An advanced programming language rewrite used in specific array operations is loop vectorization. Obviously, this optimization technique is essential in array databases. Loop vectorization consists of identifying loops that can be performed with array operations, and replace them with element-wise array operations. An example is shown below.

```
A: for (i=1:10) X[1,i] = Y[i,2]
B: X[1,1:10] = Y'[1:10,2]
```

Expression A is a loop to copy the first ten values in the second column of Y into the first ten values in the first row of X. This loop can be vectorized leading to the expression B, where the index variable `i` has been replaced by its range `1:10`. Additional changes are also required. Notice that vector Y has been transposed in order to ensure alignment for element-wise matrix operations. Nested loops can be vectorized by applying this technique repeatedly.

4.2 Physical rewrites

Avoiding unnecessary data shuffling helps to improve efficiency for large-scale ML programs. Thus, many ML systems implement different techniques for data flow rewrites. Although there are several data flow rewrite techniques (e.g., joins, aggregation types), in this section we briefly discuss two of the most important, *distributed caching* and *partitioning*.

4.2.1 Distributed caching

Deciding upon the caching of intermediates always represent a tradeoff between memory consumption and redundant computation. Existing systems use very different approaches. For example, in Mahout Samsara [10], users control caching manually at script level. All intermediates that are consumed multiple times are cached in Emma [55]. Large matrices and variables used read-only in loops are cached in SystemML [13].

4.2.2 Partitioning

A more efficient execution of joins in a distributed environment may be achieved by keeping track of partitioning. Dedicated partitioning-aware operations preserving and exploiting such partitioning are provided by systems like SystemML, Mahout Samsara, and Emma.

4.3 Data access methods

Similarly to above optimization techniques, many ML systems leverage different techniques for efficient data access that are also based on those already implemented in databases. One of the most relevant data access optimization technique is briefly discussed in this section.

4.3.1 Compression

Based on data compression and query processing over such compressed data in database systems, ML systems store tensors, matrices and graphs in compressed form, and perform linear algebra over such compressed structures. Depending on whether the compression process loses information or not, existing techniques can be grouped into lossless and lossy compression techniques.

- **Lossless Compression.** Result correctness is guaranteed with these techniques. The idea here is to use general-purpose compression techniques. This requires block-wise decompression for each operation. Unfortunately, lightweight methods provide modest compression ratios, while heavyweight techniques show very slow decompression. Some scientific data formats, like NetCDF [56], provide compression at chunk granularity, by specifying the chunk size and deflation level. Distributed-data processing frameworks, like Spark, support different storage levels and configurations for data compression.
- **Lossy Compression.** Loss of information leads to less accurate results. Thus, this techniques have to be carefully applied. However, many ML algorithms can tolerate less accuracy in results because they are approximate in nature. Tradeoff space between less accuracy and more efficiency is very wide. A good knowledge of the lossy compression impact in the ML algorithm accuracy is required. Sometimes this impact depends on input data and thus, is difficult to estimate.

5 Linear algebra operators in ML algorithms

A basic and preliminar step when trying to implement ML algorithms in Array Databases is to identify the underlying

linear algebra operations. Efficient implementations of such operations within the database will allow the efficient implementation of ML algorithms. The underlying linear algebra foundations of some common ML algorithms are provided in this section. Then, based on these mathematical constructs, the required linear algebra operations are pointed out. There are many different ML models, from basic mathematical equations to advanced ML algorithms with increasing complexity. Nevertheless, the underlying mathematical structures and operators are mostly common to all of them. Thus, these authors consider that the analysis of the linear and logistic regression algorithms (as common examples of prediction methods) and the *feed-forward* neural network (as a common example of classification method) is enough to obtain the linear algebra operators required to implement the vast majority of the remainder ML models.

5.1 Linear regression algorithm

The linear regression algorithm is one of the most used algorithms in prediction. In this section, the underlying linear algebra in linear regression is explored in order to determine the required linear algebra operations.

To introduce the appropriate notation, a motivating example for predicting house prices is going to be used. We want to predict the price of a house based on the following additional valuable information: size of the house, number of bedrooms, number of floors, and age of the house. This information variables are called *features*. Table 1 depicts some example values for this features and the corresponding output variable, i.e., the variable we want to predict. We will use this set of values to train the model, so this set is called the *training set*. We now denote the features as follows: x_1 for the size, x_2 for the number of bedrooms, x_3 for number of floors, and x_4 for the age. The output variable to be predicted is denoted by y . The number of features in the input examples is denoted by n , the number of examples in the training set is denoted by m , the i^{th} input example in the training set (i.e., the i^{th} training example) is denoted by vector $x^{(i)}$, and the value of feature j in

Table 1 Example data for predicting house prices

Size (feet ²)	Number of bedrooms	Number of floors	Age of House (years)	Price (\$1000)
2304	3	2	48	450
2004	4	1	45	385
1804	2	2	39	345
1584	2	1	35	270
1704	2	2	40	250

the i^{th} training example is denoted by $x_j^{(i)}$. For the example depicted in Table 1, we have $m = 5$, $n = 4$, $x^{(2)} = [2004, 4, 1, 45]'$ and $x_3^{(2)} = 1$.

5.1.1 Hypothesis function

The job of the learning algorithm is to output a function which is usually denoted by h , where h stands for *hypothesis*. This hypothesis is a function that takes as input the value of x and it tries to output the estimated value of y . So h is a function that maps from x 's to y 's. In our example, we represent the hypothesis function as

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \quad (1)$$

meaning that we are going to predict that y is a linear function of x . For example, $h_\theta(x) = 70 + 0.1x_1 + 0.01x_2 + 3x_3 - 2x_4$ means that the base price of a house is \$70000, plus \$100 per square feet, plus \$10 per bedroom, plus \$3000 per floor, and minus \$2000 per year the house has.

A general formulation of the hypothesis function for any number of features is as follows

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad (2)$$

For convenience of notation, we define $x_0 = 1$ for all the input examples in the training set, i.e., $x_0^{(i)} = 1$. Then, we can write vectors x and θ as follows

$$x = \begin{bmatrix} cx_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1} \quad (3)$$

Hence, now we can write our hypothesis function as

$$\begin{aligned} h_\theta(x) &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \\ &= \theta^T x \end{aligned} \quad (4)$$

5.1.2 Gradient descent

Once the hypothesis function has been defined, the algorithm needs to fit its parameters. In particular we are going to use the gradient descent for linear regression with multiple features. To provide a measure of the error in the model, a *cost* function given by the sum of the square of the error is defined as follows

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 \quad (5)$$

The gradient descent algorithm, depicted below, repeatedly updates each parameter by subtracting alpha times the partial derivative of the cost function.

Algorithm 1 Gradient descent algorithm in linear regression.

- 1: **repeat**
 - 2: $\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$
 - 3: **until** $J(\theta) < \text{max_error}$ or a specific number of iterations is reached.
-

Applying vectorization we can obtain a more efficient implementation. The vectorized version of the gradient descent is depicted below.

Algorithm 2 Vectorized Gradient descent algorithm in linear regression.

- 1: **repeat**
 - 2: $\theta := \theta - \frac{\alpha}{m} * (X^T * (X * \theta - y))$
 - 3: **until** $J(\theta) < \text{max_error}$ or a specific number of iterations is reached.
-

where

$$X = \begin{bmatrix} x_0^{(1)} & \dots & x_n^{(1)} \\ x_0^{(2)} & \dots & x_n^{(2)} \\ \vdots & & \vdots \\ x_0^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^{n+1} \quad (6)$$

Thus, from the above linear algebra expressions we can determine that the next operations are required:

- Vector/Matrix - Scalar operations: +, -, *, /.
- Vector/Matrix unary operations: transpose.
- Vector/Matrix binary operations: +, -, *, /.
- Vector/Matrix element-wise operations: .*.
- Vector/Matrix aggregate operations: sum.
- Iteration constructs.

5.2 Logistic regression algorithm

A different type of problems are the so called *classification* problems, where the output value y is either 1 or 0. We have many examples of classification problems, e.g., whether a tumor is benign or malignant, a mail is spam or not, an online transaction is fraudulent or not. Linear regression algorithms do not provide successful results when applied to classification problems. This section is devoted to *logistic regression*, one of the most popular and most widely used learning algorithms today. Logistic regression is actually a classification algorithm that we apply to settings where the label y is a discrete value, either zero or one. Contrarily to linear regression, logistic regression has the property that the outputs (predictions) are always between zero and one, i.e., $0 < h_\theta(x) < 1$.

5.2.1 Hypothesis function

As stated above, we would like our classifier to output values between 0 and 1. For logistic regression, we modify the hypothesis used in linear regression to satisfy this condition. The hypothesis for logistic regression is

$$h_{\theta}(x) = g(\theta^T x) \quad (7)$$

where g is the *sigmoid function*,

$$g(z) = \frac{1}{1 + e^{-z}} \quad (8)$$

hence,

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}} \quad (9)$$

5.2.2 Gradient descent

We could take the same approach that we used for linear regression and use a *cost* function given by the sum of the square of the error is defined as follows

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2 \quad (10)$$

But in this case, $J(\theta)$ results in a *non-convex* function. Instead, we are going to use a different cost function that is convex, so we can apply the gradient descent and be guaranteed to find the global minimum. The cost function for logistic regression is

$$Cost(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases} \quad (11)$$

The above cost function definition can be compressed into one equation

$$Cost(h_{\theta}(x), y) = -y \log(h_{\theta}(x)) - (1 - y) \log(1 - h_{\theta}(x)) \quad (12)$$

Using the above cost function, $J(\theta)$ can be calculated as follows

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^m Cost(h_{\theta}(x^{(i)}), y^{(i)}) \\ &= \frac{-1}{m} \sum_{i=1}^m y^{(i)} \log(h_{\theta}(x^{(i)})) \\ &\quad + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \end{aligned} \quad (13)$$

As we can see below, the Gradient Descent algorithm for logistic regression is exactly the same that the Gradient Descent algorithm for linear regression. The difference between them is the hypothesis function $h_{\theta}(x)$.

Algorithm 3 Gradient descent algorithm in logistic regression.

```
1: repeat
2:    $\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$ 
3: until  $J(\theta) < \text{max\_error}$  or a specific number of iterations is reached.
```

We can also apply vectorization to obtain a more efficient implementation.

Algorithm 4 Vectorized gradient descent algorithm in logistic regression.

```
1: repeat
2:    $\theta := \theta - \frac{\alpha}{m} * (X^T * (\frac{1}{1 + e^{-X * \theta}} - y))$ 
3: until  $J(\theta) < \text{max\_error}$  or a specific number of iterations is reached.
```

In addition to the operations required in linear regression, we need:

- Vector/Matrix function evaluation: $\text{sigmoid}(X * \theta)$

5.3 Feed-forward neural network

One of the most powerful learning algorithms that we have today are the neural networks. Although neural networks were originally designed to build machines for mimic the human brain, currently they are the state-of-the-art machine learning technique for many applications. Thus, based on its origins, a neural network is a group of connected neurons ordered in layers. Figure 2 depicts a neural network with four layers. The units of the first layer (input layer) store the attributes of each training example, i.e., x_1, x_2 and x_3 . Notice that a *bias unit* ($x_0 = 1$) has been added to the

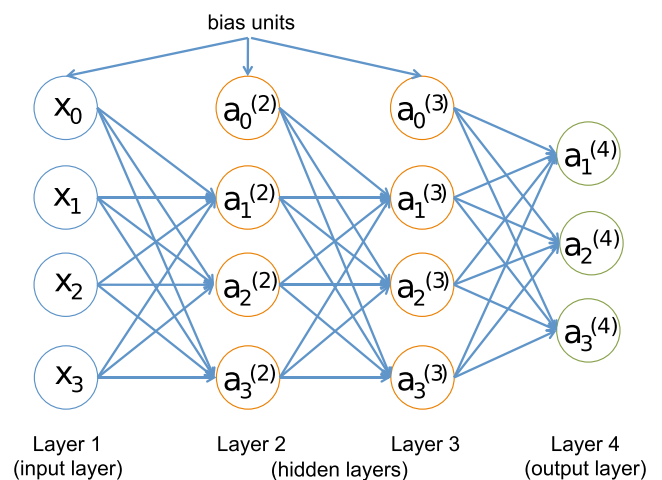


Fig. 2 Example feed-forward neural network

input layer (Layer1). The example neural networks has two hidden layers (Layer2 and Layer3) with three units plus a *bias unit* ($a_0^{(2)} = a_0^{(3)} = 1$). The *activation* of unit i in layer j is $a_i^{(j)}$. Finally, one output layer with three units provides the resulting hypothesis. Thus, this network is used to perform *multi-class* classification with three classes ($K = 3$), where

$$h_{\theta}(x) = \begin{bmatrix} a_1^{(4)} \\ a_2^{(4)} \\ a_3^{(4)} \end{bmatrix} \quad (14)$$

5.3.1 Hypothesis function

For each layer j , the network has a matrix $\theta^{(j)}$ of weights controlling the mapping from layer j to layer $j + 1$. For example, for Layer1 in the example network we have

$$\theta^{(1)} = \begin{bmatrix} \theta_{10}^{(1)} & \theta_{11}^{(1)} & \theta_{12}^{(1)} & \theta_{13}^{(1)} \\ \theta_{20}^{(1)} & \theta_{21}^{(1)} & \theta_{22}^{(1)} & \theta_{23}^{(1)} \\ \theta_{30}^{(1)} & \theta_{31}^{(1)} & \theta_{32}^{(1)} & \theta_{33}^{(1)} \end{bmatrix} \quad (15)$$

where $\theta_{xy}^{(1)}$ is the weight from unit y in layer 1 to unit x in layer 2.

To obtain the expression of the resulting hypothesis we need to perform a forward propagation from input layer to output layer in order to compute the *activation* of the units. Thus, the activation of the units in Layer2 can be computed with the expressions below.

$$\begin{aligned} a_1^{(2)} &= g(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3) \end{aligned} \quad (16)$$

where g is again the *sigmoid function*.

Similarly, we can compute the *activations* of Layer3 with the following expressions.

$$\begin{aligned} a_1^{(3)} &= g(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}) \\ a_2^{(3)} &= g(\theta_{20}^{(2)} a_0^{(2)} + \theta_{21}^{(2)} a_1^{(2)} + \theta_{22}^{(2)} a_2^{(2)} + \theta_{23}^{(2)} a_3^{(2)}) \\ a_3^{(3)} &= g(\theta_{30}^{(2)} a_0^{(2)} + \theta_{31}^{(2)} a_1^{(2)} + \theta_{32}^{(2)} a_2^{(2)} + \theta_{33}^{(2)} a_3^{(2)}) \end{aligned} \quad (17)$$

Finally, for the output layer we have,

$$\begin{aligned} a_1^{(4)} &= g(\theta_{10}^{(3)} a_0^{(3)} + \theta_{11}^{(3)} a_1^{(3)} + \theta_{12}^{(3)} a_2^{(3)} + \theta_{13}^{(3)} a_3^{(3)}) \\ a_2^{(4)} &= g(\theta_{20}^{(3)} a_0^{(3)} + \theta_{21}^{(3)} a_1^{(3)} + \theta_{22}^{(3)} a_2^{(3)} + \theta_{23}^{(3)} a_3^{(3)}) \\ a_3^{(4)} &= g(\theta_{30}^{(3)} a_0^{(3)} + \theta_{31}^{(3)} a_1^{(3)} + \theta_{32}^{(3)} a_2^{(3)} + \theta_{33}^{(3)} a_3^{(3)}) \end{aligned} \quad (18)$$

Hence,

$$h_{\theta}(x) = \begin{bmatrix} g(\theta_{10}^{(3)} a_0^{(3)} + \theta_{11}^{(3)} a_1^{(3)} + \theta_{12}^{(3)} a_2^{(3)} + \theta_{13}^{(3)} a_3^{(3)}) \\ g(\theta_{20}^{(3)} a_0^{(3)} + \theta_{21}^{(3)} a_1^{(3)} + \theta_{22}^{(3)} a_2^{(3)} + \theta_{23}^{(3)} a_3^{(3)}) \\ g(\theta_{30}^{(3)} a_0^{(3)} + \theta_{31}^{(3)} a_1^{(3)} + \theta_{32}^{(3)} a_2^{(3)} + \theta_{33}^{(3)} a_3^{(3)}) \end{bmatrix} \quad (19)$$

5.3.2 Cost function and partial derivatives

Based on exposed above and the cost function in logistic regression, we can express the cost function for the neural network as follows,

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)}) \\ &= \frac{-1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_{\theta}(x^{(i)}))_k + \\ &\quad (1 - y_k^{(i)}) \log(1 - (h_{\theta}(x^{(i)}))_k) \end{aligned} \quad (20)$$

where $y_k^{(i)}$ and $(h_{\theta}(x^{(i)}))_k$ are, respectively, the k^{th} elements of the expected output and the hypothesis output for example input $x^{(i)}$.

5.3.3 Gradient descent

In addition to the cost function, we need the partial derivatives in order to compute the gradient descent of the cost function. To compute the partial derivatives we need to execute the backpropagation algorithm. Such algorithm allows us to compute the error for each input sample and then compute the partial derivatives by backpropagating the errors from the output layer to the input layer.

First, for each training example (x, y) , the *activations* of the units can be computed by forward propagation with the vectorized implementation below,

$$\begin{aligned} a^{(1)} &= x \\ z^{(2)} &= \theta^{(1)} a^{(1)} \\ a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\ z^{(3)} &= \theta^{(2)} a^{(2)} \\ a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\ z^{(4)} &= \theta^{(3)} a^{(3)} \\ a^{(4)} &= h_{\theta}(x) = g(z^{(4)}) \end{aligned} \quad (21)$$

Once we have the resulting hypothesis, the backpropagation algorithm is used to compute the error in each unit for the training example. The errors in layer l is denoted by $\delta^{(l)}$. First, we compute the errors for the output layer. And then, we can compute the errors of the layer l by backpropagating the errors in layer $l + 1$. The vectorized implementation of the backpropagation step is as follows,

$$\begin{aligned} \delta^{(4)} &= h_{\theta}(x) - y = a^{(4)} - y \\ \delta^{(3)} &= (\theta^{(3)})^T \delta^{(4)} \cdot g'(z^{(3)}) \\ \delta^{(2)} &= (\theta^{(2)})^T \delta^{(3)} \cdot g'(z^{(2)}) \end{aligned} \quad (22)$$

When training a neural network, these errors are computed for each training example and used to compute the overall partial derivatives for the entire training set. The

algorithm of the backpropagation for the entire training set is shown below.

Algorithm 5 Backpropagation algorithm.

```

1: TrainingSet  $\leftarrow \{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ 
2:  $\Delta^{(l)} \leftarrow 0$  (for all  $i, j, l$ )
3: for  $i = 1$  to  $m$  do
4:    $a^{(1)} \leftarrow x^{(i)}$ 
5:   Compute  $a^{(l)}$  for  $l = 2, \dots, L$  (forward propagation)
6:    $\delta^{(L)} \leftarrow (a^{(L)} - y^{(i)})$ 
7:   Compute errors:  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ 
8:   Accumulate errors:  $\Delta^{(l)} \leftarrow \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$ 
9: end for
10:  $\frac{\partial J(\theta)}{\partial \theta_{ij}^{(l)}} \leftarrow \frac{1}{m} \Delta_{ij}^{(l)}$ 

```

As we can see, first the accumulated errors $\Delta^{(l)}$ are initialized to zero for all the units. Then, for each training example, the *activations* $a^{(l)}$ for all the units are computed by forward propagation. The errors $\delta^{(L)}$ for the output layer are computed next. Then, the error in the output layer is backpropagated to compute the errors $\delta^{(l)}$ in the remainder layers. The errors are accumulated in $\Delta^{(l)}$ for all the examples in the training set. Finally, we use the accumulated errors to compute the partial derivatives $\frac{\partial J(\theta)}{\partial \theta_{ij}^{(l)}}$ to be used in the gradient descent algorithm.

Thus, as we can see, the *feed-forward* neural network can be implemented with the same operations used in logistic and linear regressions algorithms. Furthermore, most of the ML algorithms can be implemented with those operators. The following Table 2 summarizes such operations.

6 Array database management systems

A detailed overview of array data and array database managers is provided in this section. Arrays, also called *raster data*, *gridded data* or *datacubes*, are core data structures that appear in almost every area of engineering and science,

Table 2 Linear Algebra Operations in ML algorithms

Vector/Matrix - Scalar operations: $+$, $-$, $*$, $/$
Vector/Matrix unary operations: transpose
Vector/Matrix binary operations: $+$, $-$, $*$, $/$
Vector/Matrix element-wise operations: $.$
Vector/Matrix aggregate operations: sum
Iteration constructs
Vector/Matrix function evaluation: sigmoid($X * \theta$)

e.g., life sciences, statistics, earth sciences, space sciences. Relational DBMSs show many problems when dealing with large binary datasets of structured information. Thus, array databases [57] have been specifically designed to solve such drawbacks. Storage and processing functionality for multi-dimensional arrays are provided by array databases through flexible and scalable services. Large sets of operations, from statistics to general Linear Algebra, build the core of such databases and provide functionality for array processing. A comprehensive survey on array databases has been recently published [58].

6.1 Array data model

Formally, a n -dimensional array is a function $f : D \rightarrow T$ which domain is the n -fold Euclidean cross product of closed integer intervals:

$$D = \{lo_1, \dots, hi_1\} \times \dots \times \{lo_n, \dots, hi_n\} \quad (23)$$

$$lo_i \leq hi_i \text{ for } 1 \leq i \leq n$$

where T is a non-empty set called the *cell type* of the array. Elements in T are called *cells*. This definition have a equivalence in mathematics where *vectors* (or *sequences*) represent 1-D arrays, *matrices* represent 2-D arrays, and *tensors* represents higher-dimensional arrays. The core operations on arrays have been defined by Tomlin in the well known Map Algebra. This operations have been categorized depending of the number of cells of an input array that are taken into account to compute each cell of the resulting array.

6.2 Array databases

For decades, relational DBMSs principles have been proven successful. Now, array databases are offering their benefits as well.

- **Declarative query language.** Rather than write a procedural algorithm to obtain the desired result, users provide a description of such result. Thus, an array query written in 2-3 lines of code would be translated into several pages of procedural code.
- **Transparent storage management.** Although many data centers are used to know the exact location of each byte on disk, this transparency has two important advantages: (1) the user access is more simple, and (2) internal reorganizations may be performed without affecting users.
- **Concurrency and access control.** Access management to large amounts of data that are usually queried by many users simultaneously is required. Inconsistencies due to parallel accesses to data must be avoided by

concurrency control systems. Specifically for arrays, granularity of access control must allow access management of arbitrary areas within datacubes.

6.2.1 Storage

The Euclidean neighborhood of array cells is a key concept when implementing access patterns on arrays. Any storage engine must implement some spatial clustering strategy to preserve proximity on persistent storage. It is a common practice to partition n-D arrays into n-D subarrays, also called tiles [59] or chunks [60], to form the unit of access to persistent storage. Given the wide variety of possible workloads, the challenge here is to find the partitioning technique that minimizes the number of partitions fetched from persistent storage and, hence, the query performance. Although this principle is generally accepted, there are multiple variations for partitioning techniques.

Spatial indexes (e.g., R-Tree) have proven advantageous to determine the required partitions much faster. Most spatial indexes show good performance because the targeted objects, which have a box structure, partition a space of known extent. Contrarily, spatial databases manage polygons which lack that regular structure. Additionally, another advantageous technique is compression of tiles.

6.2.2 Processing

In general, relational query processing is typically I/O bound. On the contrary, query evaluation in array operations is heavily CPU bound. Many array operations are trivially parallelizable and, thus, easily distributed both in local nodes (general-purpose GPUs and multicore CPUs) and remote nodes (clusters, cloud networks). Some other operations require to be transformed, and sometimes rewritten (Fig. 3) in a new set of operations, before being parallelized, e.g., histogram generators, joins of differently partitioned arrays. Next, some optimizations proven effective in array DBMSs are briefly explained.

- **Parallelization.** Sometimes array operations have been considered “embarrassingly parallel” due to (1) array operations apply the same operation over a large number of values, and (2) tiles map naturally to CPU cores. This is only true for simple operations, but not true in general. Even simple binary operations like $a + b$ may be challenging, e.g., both operands are in different nodes and have incompatible tiling. Additional complexity may be found in Linear Algebra operations. Several criteria for splitting queries across multiple systems may be followed, e.g., data location, intermediate results transfer costs, current resource availability.

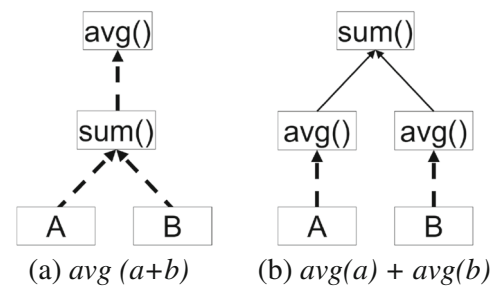


Fig. 3 Equivalence rule for array query rewriting: “ $avg(a + b) \equiv avg(a) + avg(b)$ ”

- **Mixed Hardware.** Although processing time can be highly speeded up by compiling queries into code for CPUs, GPUs and FPGAs, some non-trivial problems on mixed hardware evaluation are still under active research.
- **Approximate Caching.** Similarly to RDBMSs, caching final or intermediate array results can improve efficiency when same or similar queries are frequently executed. For example, mitigation forces and general public will issue lots of queries on a disaster region. With array data we may find that these queries do not hit exactly the same area, but they may share partially the requested region. So, the query engine can also reuse partially matching areas in arrays to provide faster results [61].
- **Cost-based Optimization.** Similarly to RDBMSs, this optimization techniques attempts to find the most efficient execution plan among all possible plans for a given query. Contrarily to query rewriting, knowledge of the processing costs is required. Many parameters can influence such costs, e.g., location of tiles (distributed environments), number of tiles to be read from persistent storage, number and complexity of operations.

6.3 Classification

Due to the blooming research and development experienced in this area, further systems not referenced in this survey may emerge soon. Systems encountered in literature may be grouped into four main categories.

- **Array Database Systems.** Relevant features include multi-user operation, access control mechanisms, storage management and query language. These systems may be subdivided into two groups.
 - **Full-stack Array Databases.** Systems implemented from the scratch, e.g., rasdaman [51], SciDB [52].

- **Add-ons to existing database systems.** May be implemented by adding extra layers to existing DBMSs (e.g., EXTASCID [62]), performing direct DBMS kernel coding (e.g., SciQL [63]), or providing object-relational extensions (e.g., PostGIS Raster [64], Teradata Arrays [65], OracleGeoRaster [66]).
- **Array Tools.** Systems encompassing libraries and command-line tools that do not constitute a complete service tool but may be components of larger array services. Since data scientists generally prefer high-level languages, these solutions are preferably used in data centers where data experts are also experienced full-stack developers.
- **MapReduce Array Engines.** Systems providing processing capabilities for multi-dimensional arrays on top of Hadoop or Spark frameworks.

6.4 Full-stack array DBMSs

6.4.1 Rasdaman

Since its first publication in 1992, rasdaman [51] has pioneered the field of array databases. Extraction, retrieval, fusion and aggregation on distributed arrays is performed in the server through parallelization, effective optimization and heterogeneous hardware. Single point of failure is avoided in rasdaman by using a parallelizing peer federation architecture. Arrays can be stored either in the optimized array store or in standard databases. Moreover, any pre-existing archive structure can be used. A SQL-based query language, called rasQL, is provided in its query interface. Rasdaman is also a pattern for several Big Data standards, such as the ISO SQL/MDA (Multi-Dimensional Arrays) [67] and the OGC Web Coverage Service (WCS) with its geo datacube query language, Web Coverage Processing Service (WCPS) [68].

6.4.2 SciDB

Similarly to rasdaman, SciDB [52] is an array DBMS that provides an Array Query Language (AQL) and an Array Functional Language (AFL). A modified Postgres kernel is the basis for its architecture, plus UDFs implementing array functionality.

6.5 Extensions to existing DBMSs

6.5.1 SciQL

SciQL [63] was born as an extension of the column-store DBMS MonetDB with array-specific operators. There is no dedicated storage and processing engine in SciQL, n-D

arrays are internally sequentialized and stored into column-store tables.

6.5.2 EXTASCID

Built as a complete and extensible system for scientific data processing for both arrays and relational data, EXTASCID [62] considers the execution of arbitrary user code as a central part of its design. As a result, supports in-database processing of full scientific workflows over both raw and derived data. EXTASCID is built around the massively parallel GLADE architecture for data aggregation and enhances the original GLA interface implemented in GLADE with functions specific to scientific processing.

6.5.3 PostGIS Raster

Geo raster data can be stored and analyzed using the PostGIS type “Raster” [64]. While the implementation of this type heavily relies on the extension capabilities of PostgreSQL, GDAL is used for processing purposes. Although raster expressions are allowed, they are not integrated with the PostgreSQL query language. Instead, they are written in a separate Map Algebra language and passed to a raster object as a string. Large objects are not partitioned automatically, users have to perform such partition. Then, partitions are distributed over tuples in raster columns. Hence, queries have to be written in such a way that a proper recombination of large rasters from partitions stored in one tuple each can be achieved.

6.5.4 Oracle GeoRaster

Both spatial data types and an object-relational schema are provided by Oracle GeoRaster [66]. Digital images and multi-dimensional grid layers referencing either a local coordinate system or locations on the Earth’s surface can be stored by using these elements. Neither a specific raster query language nor a specific array-centric architecture are provided.

6.5.5 Teradata arrays

Similarly to Oracle GeoRaster, Teradata has recently proposed an object-relational solution to add arrays as a new datatype [65]. Each array is mapped to 64kB blobs including the metadata, providing a size array of 40 kB approximately.

6.6 Array tools

6.6.1 wendelin.core

Wendelin.core [69] allows for processing of big arrays, bigger than RAM and local disk, which can be processed

with transactions and persisted in storage. The virtual address-space is the limit for the big array size.

6.6.2 TensorFlow

TensorFlow [70] is a well known tool for machine learning, mainly focused on deep neural networks, that provides a wide range of functionalities

6.6.3 xtensor

The Python array programming library NumPy is used in xtensor [71] to build a C++ library for numerical analysis. Although multi-dimensional array expressions can be defined, an extensible expression system is provided. Thus, lazy broadcasting, tools for array expression manipulation and an API following the C++ standard library are enabled.

6.6.4 OPeN-DAP

The Open-source Project for a Network Data Access Protocol (OPeN-DAP) [72] is both a transport protocol and an data architecture for earth scientists. Standards for encapsulation of structured data, addition of semantics describing such data, and annotation of data with attributes are defined. Arrays are one-dimensional. Multi-dimensional arrays are defined as arrays of arrays. Constraint expressions allow clients to request variables or parts of variables.

6.6.5 Ophidia

A full software stack enabling data analytics and management of big datasets is provided by Ophidia [73]. Server-side approach, hierarchically distributed storage, and parallel and in-memory computation are main features in this system. Processing of multi-dimensional array-based data is supported by the implementation of the datacube abstraction in its data model. A wide set of operations is provided for data analytics.

6.6.6 Google earth engine

Google Earth Engine [74] relies on grid systems with files. With a functional programming language in its heart, code can be submitted to Google Earth Engine by users and executed in Google's own distributed environment.

6.6.7 TileDB

A library for management of sparse and dense arrays is provided by TileDB [75]. Any number of attributes of

different data types can be stored in each multi-dimensional array in TileDB. High I/O on several data persistence backends, compression and integration with well known data science ecosystems are relevant features on this library.

6.6.8 boost::geometry

Primitives and algorithms for solving geometry problems are defined in this boost.Geometry [76]. A generic definition of a multi-dimensional array is provided by the interface *Boost.MultiArray* interface. Some common implementations of this interface are also defined.

6.6.9 Open DataCube

A free and accessible exploitation architecture is provided by the Open Data Cube (ODC) initiative [77] to boost the impact of satellite Earth observation data.

6.6.10 Xarray

Xarray [78] is an open source project which aims to provide a pandas-like [79] toolkit to support analytics on multi-dimensional arrays.

7 ML in Array DBMSs

7.1 System comparison

The importance of several database concepts, such as partitioning, caching and compression, applied to the efficient processing of algorithms in ML systems was stated in Section 4. This section provides an overview of the implementation state of those concepts in the above databases. Tables 3 and 4 summarize the information detailed below.

7.2 Partitioning

When size of arrays is larger than server RAM or disk partitions, partitioning becomes indispensable for handling them. SciQL, Teradata Arrays, TensorFlow, Ophidia, Google Earth Engine, boost::geometry, Open Data Cube, and xarray do not support array partitioning. Rasdaman and EXTAS-CID provide full support for multi-dimensional arrays with any tiling. Some solutions provide limited support, PostGIS Raster supports partitioning only for small arrays, Oracle GeoRaster only performs partitioning during raster creation, SciDB only implements partitioning for regular multi-dimensional arrays, OPeN-DAP provides partitioning per NetCDF file, and TileDB supports regular tiling.

Table 3 Database features in ML

Features	Full-stack Array DBMS		Array DBMS Add-on array support				
	Rasdaman [51]	SciDB [52]	SciQL [63]	EXTASCID [62]	PostGIS Raster [64]	Oracle GeoRaster [66]	Teradata Arrays [65]
Partitioning	any nD tiling	regular nD	no	any nD chunking	small arrays (100x100)	yes (during raster creation)	no
Compression	lossy and lossless (zlib, RLE, wavelets,...)	RLE	no	no	no	yes (JPEG, DEFLATE)	no
Distribution	automatic query distribution, peer federation (shared nothing)	shared nothing	no	shared memory, shared disk servers, shared nothing	no	yes	no
Caching	yes (can reuse approximate caching)	yes (persistent chunk caching, temporary result caching)	?	no	no	yes	no
Query rewriting	yes	yes	yes	no	no	no	no
Common subexpression elimination	yes	?	?	no	no	no	no
Cost-based optimization	yes	?	?	no	no	no	no

Table 4 Database features in ML

Features	wendelin.core	TensorFlow	Open-DAP	Array Tools		TileDB	boost::geometry	Open Data Cube	xarray
				Ophidia	Google Earth Engine				
	[69]	[70]	[72]	[73]	[74]	[75]	[76]	[77]	[79]
Partitioning	maybe indirectly	no	per NetCDF	no	no	regular tiling	no	no	no
Compression	no	sparse tensor	per NetCDF	zlib	no	per tile	no	no	no
Distribution	maybe indirectly	Cloud ML	no	yes	no	only if VFS supports it	no	no	no
Caching	yes	yes	no	?	yes	yes	no	yes	no
Query rewriting	no	no	no	no	no	no	no	no	no
Common subexpression elimination	no	no	no	no	yes	no	no	no	no
Cost-based optimization	no	no	no	no	yes	no	no	no	no

Implementation on array Tools

7.3 Compression

Both lossy and lossless compression techniques are included. The impact of lossless compression is highly dependent on the data properties, e.g., natural images can compress to about 80% whereas thematic map layers can compress to about 5%. Lossy compression is also used in some systems but it may introduce inaccuracies in tile boundaries. Both lossy and lossless compression schemas are offered by Rasdaman, e.g., Zlib, RLE, wavelets. Some systems do not provide compression at all, e.g., SciQL, EXTASCID, PostGIS Raster, Teradata Arrays, wendelin.core, Google Earth Engine, boost::geometry, Open Data Cube, and xarray. JPEG and DEFLATE schemas are provided by Oracle GeoRaster, whereas SciDB only supports run length encoding (RLE). TensorFlow provide compression support for sparse tensors. OPeN-DAP also provides compression per NetCDF file. Zlib is supported in Ophidia. TileDB supports compression per tile.

7.4 Distribution

Horizontal scaling is enabled when either complete arrays or tiles of an array can be partitioned. Of course, this requires from dynamic reassembly which have to be performed carefully to ensure a satisfying performance in computationally expensive operations. SciQL, PostGIS Raster, Teradata Arrays OPeN-DAP, Google Earth Engine, boost::geometry, Open Data Cube, and xarray do not provide support for array distribution. A *shared nothing* strategy is provided by SciDB. Rasdaman offers automatic query distribution in a peer federation environment with a *shared nothing* strategy as well. EXTASCID supports different options: *shared*

nothing, *shared memory* and *shared disk*. TensorFlow provide support for distribution on Cloud ML. TileDB relies on the underlying VFS to support distribution.

7.5 Caching

A significant speed-up can be accomplished by caching. Some relevant questions arise when implementing caching strategies, e.g., should intermediate results be cached or only complete results?, can we reuse approximate cache hits or exact cache content is required?. Rasdaman, SciDB, Oracle Raster, wendelin.core, TensorFlow, Google Earth Engine, TileDB, and Open Data Cube provide caching capabilities. While Rasdaman also allows for the reuse of approximate cached results, SciDB implements both persistent chunk caching and temporary chunk caching.

7.6 Query rewriting

This technique has two main advantages: (1) replacing query expressions by more efficient versions can have a significant impact, as shown in Section 6, and (2) users are free from providing the most efficient formulation. Both a query language and runtime analysis of incoming code are required to implement this feature. Query rewriting is supported by Rasdaman, SciDB and SciQL.

7.7 Common subexpression elimination

Query engines implementing this technique are able to identify identical parts within a query, appearing several times in the query execution tree, and evaluate them only

once. Similarly to query rewriting, this frees users from write their queries in the most efficient way. Support for common subexpression elimination has been documented only for Rasdaman and Google Earth Engine.

7.8 Cost-based optimization

As already stated in Section 6, cost-based optimization tries to find the most efficient expression taking into account information about the execution costs of operators, rather than following pre-defined rules. For illustration purposes, let us think in an join operation between array A and array B, stored in execution nodes A and B respectively. Depending on whether array A is copied to the node B, or array B is copied to the node A, or a shared approach is implemented, a significant difference in the execution time may occur. Many impact factors (e.g., the actual tiling of both arrays) can be taken into account in the decision making. Only Rasdaman and Google Earth Engine have documented this optimization feature.

7.9 ML implementations

To the best of these authors knowledge, only Rasdaman and SciDB are able to implement ML algorithms. This section explores these two different approaches, providing a deeper view of ML implementation in both systems, through two examples of ML algorithms implemented in both systems.

```

rasql -q 'SELECT encode(
  ml.linear_regression(c[274:312,1238,380], 37, 1500, 0.01, 0.0),
  "csv")
from S1_windspeed_w as c' --out string

```

Code 10 rasQL query.

where c is a stored coverage, the number of examples in the training set is 37, the number of iterations of the gradient

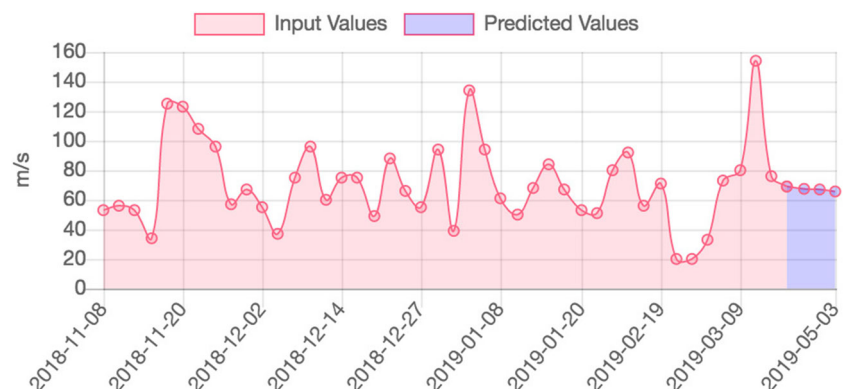
7.10 Rasdaman

Since its first publication in 1992, rasdaman has pioneered the field of array databases. Extraction, retrieval, fusion and aggregation on distributed arrays is performed in the server through parallelization, effective optimization and heterogeneous hardware. Arrays can be stored either in the optimized array store or in standard databases. A SQL-based query language, called rasQL, is also provided. Rasdaman is also a pattern for several Big Data standards, such as the ISO SQL/MDA (Multi-Dimensional Arrays) [67] and the OGC Web Coverage Service (WCS) with its geo datacube query language, Web Coverage Processing Service (WCPS).

UDTs and UDFs allow the use of linear algebra and machine learning specific libraries. However, this could lead to a decrease in the efficiency of machine learning algorithms since different database optimization strategies can not be applied. On the contrary, these optimization strategies might be applied in the case of the implementation of those algorithms in the database. Currently, the implementation of several machine learning algorithms can be done in Rasdaman through UDFs implementing the underlying linear algebra operations directly over the arrays.

Since machine learning algorithms can be implemented as UDFs within the system, they can be called in rasQL queries. Thus, users only need to call them with appropriate parameters in a rasQL query. An example rasQL query to train a simple univariate gradient descent linear regression algorithm with regularization is shown below.

Fig. 4 Training (red) and predicted (blue) wind speed values for the linear regression algorithm



The code of the user-defined function `linear_regression`, called by the above `rasQL` query, is shown below.

```
LinearRegression::LinearRegression(double *x, double *y, long num_iter,
    long num_examples, double alpha, double lambda){

    long i, count;
    theta[0] = 0.0;
    theta[1] = 0.0;
    double k = 0.0;
    double grad[2];

    if (num_examples > 0L){
        k = alpha / num_examples;

        for (i = 0L; i < num_iter; i++){
            grad[0] = 0.0;
            grad[1] = 0.0;

            for (count = 0L; count < num_examples; count++){
                double interm = (1 * theta[0] + x[count] * theta[1]) - y[count];
                grad[0] = grad[0] + interm;
                grad[1] = grad[1] + (interm * x[count]);
            }

            theta[0] = theta[0] - k * grad[0];
            theta[1] = theta[1] - k * grad[1] - (k * lambda * theta[1]);
        }
    }
}
```

Code 11 User-Defined Linear Regression function.

Note that computations in this algorithm are applied directly on the input training arrays `x` and `y`. An intermediate pre-processing step obtains the input arrays `x` and `y` from the coverage `c` provided by the user `rasQL` query.

By using the Web Coverage Processing Service (WCPS) [80] exposed by Rasdaman we can send queries to the Rasdaman database. An example WCPS query that predicts the wind speed at a specific georeferenced point for the requested dates using a linear regression algorithm trained with wind speeds of previous dates selected by the user is shown below.

```
for c in (S1_windspeed_w)
return encode(
    ml.linear_regression(
        c[ansi("2018-11-08":"2019-03-27")],
        Lat(53.866278),
        Long(8.304296)],
        "2019-03-28,2019-04-05,2019-05-03"
    ),
    "csv")
```

Code 12 WCPS query.

where

- `c` iterates over all the coverages stored in the array `datacube S1_windspeed_w`.

- `"2018-11-08":"2019-03-27"` is the set of previous dates selected to train the model.
- `(53.866278,8.304296)` and `"2019-03-28, 2019-04-05, 2019-05-03"` are, respectively, the georeferenced point and the dates for which the user want to predict the wind speed.

Figure 4 depicts the predicted values for this example together with training values. Predicted values are colored in blue and training values are colored in red.

7.11 SciDB

Similarly to `rasdaman`, `SciDB` is an array DBMS that provides an Array Query Language (AQL) and an Array Functional Language (AFL). A modified Postgres kernel is the basis for its architecture, plus UDFs implementing array functionality. Users need to use the provided linear algebra operators as building blocks to implement the ML algorithms. One of the most basic and used ML algorithm for classification is the logistic regression algorithm shown in Algorithm 6. In this case, the iterative optimization algorithm is the Batch Gradient Descent (BGD).

Algorithm 6 Logistic regression with BGD.

```

1:  $\mathbf{X} \leftarrow$  features (training dataset)
2:  $\mathbf{y} \leftarrow$  labels (training dataset)
3:  $I \leftarrow$  number of iterations
4:  $\alpha \leftarrow$  step size
5: for  $i = 1$  to  $I$  do
6:    $\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \mathbf{X}^T \left( \frac{1}{1+e^{-\mathbf{X}\mathbf{w}}} - \mathbf{y} \right)$ 
7: end for

```

The example code to implement the above logistic regression algorithm in SciDB, available for download¹, is shown below.

```

def logit(X_table_name, y_table_name, cxn, iterations=3):
    logit_iter = """
    store(
      project(
        apply(
          join(
            w,
            project(
              apply(
                gemm(
                  {X},
                  project(
                    apply(
                      join(gemm({X}, w, ZN), {Y}),
                      delta,
                      val - pow(1+EXP(-gemm), -1)),
                      delta),
                    ZK,
                    transa:true),
                    norm,
                    0.001*gemm/{N}),
                    norm)
              ),
            update,
            val+norm),
            update),
            w_new)
    """.format(X=X_table_name, Y=y_table_name, N=N)

    iteration = 0
    while iteration < iterations:
        print 'Iteration:{}'.format(iteration)
        cxn.iquery(logit_iter)
        cxn.iquery('remove(w)')
        cxn.iquery('store(project(apply(w_new, val, update), val), w)')
        iteration += 1

```

Code 13 Built-in Linear Regression function.

The outer loop of the BGD algorithm is implemented in Python because SciDB lacks form iterative constructs. For

each iteration, a SciDB query is first executed to calculate the partial resulting model and the final resulting model is updated. Notice that the code below must be implemented by users. SciDB only provides the basic linear algebra operand for matrix multiplication (gemm).

¹[https://github.com/ADALabUCSD/SLAB/blob/master/tests/MLAlgorithms%20\(Single%20Node%20Dense\)/src/scidb_algs.py](https://github.com/ADALabUCSD/SLAB/blob/master/tests/MLAlgorithms%20(Single%20Node%20Dense)/src/scidb_algs.py)

8 Conclusions

Different strategies towards the integration of ML and database systems have been surveyed in this paper. Special emphasis has been devoted to the integration of ML algorithms and arrays databases.

We have seen how linear algebra operations can be implemented in relational databases. From simple straightforward SQL implementations to more elaborated approaches, e.g., by using user-defined functions. Iterative algorithms can also be implemented in current relational databases by using simple user-defined aggregates.

More sophisticated solutions go beyond the simple SQL implementation and aim to leverage the underlying relational data model. Two main approaches were overviewed. *Learning over join* leverages database dependencies to accelerate ML algorithms. Statistical relational learning uses the structure information in the underlying data model to learn over multiple-table datasets without making the IID assumption.

The list of linear algebra operations that array databases should efficiently implement has been provided. An exhaustive study of the linear algebra foundations of four core ML learning algorithms has been undertaken.

Previous approaches are devoted to relational databases, but many ML algorithms use different types of data. Thus, several systems have been developed to process large amounts of non-relational data. Specifically, those devoted to large arrays of data, which are the main focus of this paper.

Then, a brief description of the most relevant DB-based optimization techniques used in ML systems was provided. Both logical and physical optimization rewrites were exposed, e.g., common subexpression elimination, loop vectorization, caching, partitioning. Moreover, data access techniques in databases have been also adapted to be used in ML systems to improve efficiency in ML algorithms. Both lossy and lossless compression techniques were discussed.

Beginning with the array data model, current features of array databases have been described. Depending on their architecture, array databases may be grouped into four categories. A brief description of the most relevant frameworks for array data processing in each category was provided. For each category, a detailed discussion on the implementation of the aforementioned optimization techniques was pointed out. A summary is provided below.

- Except for TensorFlow, Array Tools do not provide ML capabilities at all. Although TensorFlow is widely used in ML, it has some drawbacks, e.g., it does not provide Partitioning, Query rewriting and Cost-based Optimization.
- Xtensor, Ophidia and Xarray are the Array Tools that provide the underlying mathematical operations required to implement ML algorithms. However, none of them

provide built-in ML functionality. Moreover, Xarray do not provided any of the database features in ML, and Ophidia only provides Compression and Distribution.

- None of the Array Add-ons provide ML capability.
- Only EXTASCID and PostGIS Raster provide the required mathematical operations for ML implementation. However, PostGIS Raster only provides Partitioning for small arrays, while EXTASCID provides Partitioning and Distribution.
- Regarding full-stack databases, both Rasdaman and SciDB provide the underlying mathematical operators but none of them provide built-in implementation of ML algorithms. Rasdaman provides full implementation of the required database features in ML, while SciDB provide limited functionality of some features, e.g., Partitioning, Compression and Distribution.

Finally, two examples of ML algorithm implementation have been described. As stated above, only those systems categorized as full-stack array databases provide full support to implement ML algorithms but do not provide built-in ML algorithms. Thus, the implementation of more complex machine learning algorithms, such as deep learning methods, requires a large implementation effort from the users of such databases.

Final judgments on machine learning in array databases are stated below.

- Much research effort has to be undertaken to raise the ML support in array databases to the level of ML support in relational databases.
- TensorFlow is the only array tool currently used to implement ML algorithms. However, the lack of some database features makes the execution of models with very large matrices (data matrix and/or model matrix) very slow and sometimes unfeasible.
- None of the remainder array tools nor the array add-ons provide the database features required for the efficient implementation of ML algorithms.
- In our opinion, the most appropriate tools to provide built-in implementation of complex ML algorithms are those categorized as full-stack array databases, since only they provide all the necessary features and required mathematical operations. However, as shown in Section 7.9, none of them currently provide such functionality and users have to implement ML algorithms from the scratch.

9 Future work

Several issues arise when designing and implementing ML algorithms in array databases. A key feature that allows

array databases to achieve good performance is related to the efficient implementation of linear algebra operations on distributed array data. Current ML algorithms may have to perform computations on large matrices because both the training dataset matrix and the model matrix may be very large. When these matrices do not fit in main memory, both data and model need to be distributed. Fortunately, array databases can model these matrices as distributed arrays and provide efficient storage techniques. However, the efficient implementation of linear algebra operations (e.g. matrix multiplication) involving such distributed arrays still remains as a challenging problem which deserves research efforts.

One of the most important research topics is related to the declarative specification of iterative algorithms. Currently, many array databases provide SQL-based query languages but do not provide constructs for implementing *For loops*. But even if they were provided, their execution would not be efficient. Thus, constructs for declarative specification of such iteration algorithms in provided query languages is required.

Related to the above, declarative specification of iterative algorithms give rise to very large execution plans. How array databases can compile and execute these large and complex computations is also a challenging research topic.

Acknowledgements The work in this paper is part of the DeepRain project, funded by the German ministry of education and research (BMBF) under grant number 01 IS 18047.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Kim M, Candan KS (2014) TensorDB: In-database tensor manipulation with tensor-relational query plans. In: Proceedings of the 23rd ACM International conference on conference on information and knowledge management. CIKM '14, pp 2039–2041. ACM. <https://doi.org/10.1145/2661829.2661842>
- Cohen J, Dolan B, Dunlap M, Hellerstein JM, Welton C (2009) MAD skills: New analysis practices for big data. Proc VLDB Endow 2(2):1481–1492. <https://doi.org/10.14778/1687553.1687576>
- Feng X, Kumar A, Recht B, Ré C (2012) Towards a unified architecture for in-RDBMS analytics. In: Proceedings of the 2012 ACM SIGMOD International conference on management of data. SIGMOD '12, pp 325–336. ACM. <https://doi.org/10.1145/2213836.2213874>
- Zhang Y, Zhang W, Yang J (2010) I/O-efficient statistical computing with RIOT. 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), pp 1157–1160
- Luo S, Gao ZJ, Gubanov M, Perez LL, Jermaine C (2018) Scalable linear algebra on a relational database system. SIGMOD Rec 47(1):24–31. <https://doi.org/10.1145/3277006.3277013>
- Hellerstein JM, Ré C, Schoppmann F, Wang DZ, Fratkin E, Gorajek A, Ng KS, Welton C, Feng X, Li K, Kumar A (2012) The MADlib analytics library: Or MAD skills, the SQL. Proc VLDB Endow 5(12):1700–1711. <https://doi.org/10.14778/2367502.2367510>
- Cheng Y, Qin C, Rusu F (2012) GLADE: Big data analytics made easy. In: Proceedings of the 2012 ACM SIGMOD International conference on management of data. SIGMOD '12, pp 697–700. ACM. <https://doi.org/10.1145/2213836.2213936>
- D'silva JV, De Moor F, Kemme B (2018) AIDA: Abstraction for advanced in-database analytics. Proc VLDB Endow 11(1):1400–1413. <https://doi.org/10.14778/3236187.3236194>
- Deshpande A, Madden S (2006) MauveDB: Supporting model-based user views in database systems. In: Proceedings of the 2006 ACM SIGMOD International conference on management of data. SIGMOD '06, pp 73–84. ACM. <https://doi.org/10.1145/1142473.1142483>
- Schelter S, Palumbo A, Quinn S, Marthi S, Musselman A (2016) Samsara: Declarative machine learning on distributed dataflow systems. In: NIPS MLSSy's workshop, pp 1–8
- Sujeeth AK, Lee H, Brown KJ, Chafi H, Wu M, Atreya AR, Olukotun K, Rompf T, Odersky M (2011) OptiML: An implicitly parallel domain-specific language for machine learning. In: Proceedings of the 28th International conference on international conference on machine learning. ICML '11, pp 609–616. Omnipress. <http://dl.acm.org/citation.cfm?id=3104482.3104559>. Accessed 12 Oct 2019
- Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on operating systems design and implementation. OSDI'16, pp 265–283. USENIX Association. <http://dl.acm.org/citation.cfm?id=3026877.3026899>. Accessed 12 Oct 2019
- Boehm M, Dusenberry MW, Eriksson D, Evfimievski AV, Manshadi FM, Pansare N, Reinwald B, Reiss FR, Sen P, Surve AC, Tatikonda S (2016) SystemML: Declarative machine learning on Spark. Proc VLDB Endow 9(13):1425–1436. <https://doi.org/10.14778/3007263.3007279>
- Park Y, Qing J, Shen X, Mozafari B (2019) BlinkML: Efficient maximum likelihood estimation with probabilistic guarantees. In: Proceedings of the 2019 International conference on management of data. SIGMOD '19, pp 1135–1152. ACM. <https://doi.org/10.1145/3299869.3300077>
- Yu Y, Tang M, Aref WG, Malluhi QM, Abbas MM, Ouzani M (2017) In-memory distributed matrix computation processing and optimization. In: 2017 IEEE 33rd International conference on data engineering (ICDE), pp 1047–1058. <https://doi.org/10.1109/ICDE.2017.150>
- Bosagh Zadeh R, Meng X, Ulanov A, Yavuz B, Pu L, Venkataraman S, Sparks E, Staple A, Zaharia M (2016) Matrix computations and optimization in Apache Spark. In: Proceedings of the 22nd ACM SIGKDD International conference

- on knowledge discovery and data mining. KDD '16, pp 31–38. ACM. <https://doi.org/10.1145/2939672.2939675>
17. Villarroya S, Baumann P (2020) On the integration of machine learning and array databases. In: 2020 IEEE 36th International conference on data engineering (ICDE), pp 1786–1789. IEEE Computer Society. <https://doi.org/10.1109/ICDE48307.2020.00170>
 18. Rodrigues Zalipynis RA (2021) Towards machine learning in distributed array DBMS : Networking considerations. In: Renault, é., Boumerdassi, S., Mühlethaler, P. (eds.) Machine Learning for Networking, pp 284–304
 19. Ordoñez C, Zhang Y, Johnsson SL (2019) Scalable machine learning computing a data summarization matrix with a parallel array DBMS. Distributed and Parallel Databases 37:329–350. <https://doi.org/10.1007/s10619-018-7229-1>
 20. Baxter J (2000) A model of inductive bias learning. J Artif Int Res 12(1):149–198
 21. Caruana R (1993) Multitask learning: a knowledge-based source of inductive bias. In: Proceedings of the 10th International conference on international conference on machine learning. ICML'93, pp 41–48. Morgan Kaufmann Publishers Inc. <http://dl.acm.org/citation.cfm?id=3091529.3091535>. Accessed 12 Oct 2019
 22. Faghmous JH, Le M, Uluyol M, Kumar V, Chatterjee S (2013) A parameter-free spatio-temporal pattern mining model to catalog global ocean dynamics. In: 2013 IEEE 13th International conference on data mining, pp 151–160. <https://doi.org/10.1109/ICDM.2013.162>
 23. Liu Y, Bahadori MT, Li H (2012) Sparse-GEV: Sparse latent space model for multivariate extreme value time series modeling. In: Proceedings of the 29th international conference on international conference on machine learning. ICML'12, pp 1195–1202. Omnipress. <http://dl.acm.org/citation.cfm?id=3042573.3042727>. Accessed 12 Oct 2019
 24. Becker AS, Marcon M, Ghafoor S, Wurnig MC, Frauenfelder T, Boss A (2017) Deep learning in mammography: Diagnostic accuracy of a multipurpose image analysis software in the detection of breast cancer. Invest Radiol 52(7):434–440
 25. Liu S, Liu S, Cai W, Pujol S, Kikinis R, Feng D (2014) Early diagnosis of Alzheimer's disease with deep learning. In: 2014 IEEE 11th International symposium on biomedical imaging (ISBI), pp 1015–1018. <https://doi.org/10.1109/ISBI.2014.6868045>
 26. Lee H, Tajmir S, Lee J, Zissen M, Yeshiwas BA, Alkasab TK, Choy G, Do S (2017) Fully automated deep learning system for bone age assessment. J Digital Imaging 30(4):427–441. <https://doi.org/10.1007/s10278-017-9955-8>
 27. Kallenberg M, Petersen K, Nielsen M, Ng AY, Diao P, Igel C, Vachon CM, Holland K, Winkel RR, Karssemeijer N, Lillholm M (2016) Unsupervised deep learning applied to breast density segmentation and mammographic risk scoring. IEEE Trans Med Imaging 35(5):1322–1331. <https://doi.org/10.1109/TMI.2016.2532122>
 28. Boehm M, Kumar A, Yang J (2019) Data management in machine learning systems. Synthesis Lectures on Data Management 14(1):1–173. <https://doi.org/10.2200/S00895ED1V01Y201901D TM057>
 29. Jankov D, Luo S, Yuan B, Cai Z, Zou J, Jermaine C, Gao ZJ (2019) Declarative recursive computation on an RDBMS: Or, why you should use a database for distributed machine learning. Proc VLDB Endow 12(7):822–835. <https://doi.org/10.14778/3317315.3317323>
 30. Kumar A, Naughton J, Patel JM (2015) Learning generalized linear models over normalized data. In: Proceedings of the 2015 ACM SIGMOD International conference on management of data. SIGMOD '15, pp 1969–1984. ACM. <https://doi.org/10.1145/2723372.2723713>
 31. Schleich M, Olteanu D, Ciucanu R (2016) Learning linear regression models over factorized joins. In: Proceedings of the 2016 International conference on management of data. SIGMOD '16, pp 3–18. ACM, New York, NY, USA. <https://doi.org/10.1145/2882903.2882939>
 32. Nikolic M, Olteanu D (2018) Incremental view maintenance with triple lock factorization benefits. In: Proceedings of the 2018 International conference on management of data. SIGMOD '18, pp 365–380. ACM. <https://doi.org/10.1145/3183713.3183758>
 33. Rendle S (2013) Scaling factorization machines to relational data. Proc VLDB Endow 6(5):337–348. <https://doi.org/10.14778/2535573.2488340>
 34. Kumar A, Jalal M, Yan B, Naughton J, Patel JM (2015) Demonstration of santoku: Optimizing machine learning over normalized data. Proc VLDB Endow 8(12):1864–1867. <https://doi.org/10.14778/2824032.2824087>
 35. Chen L, Kumar A, Naughton J, Patel JM (2017) Towards linear algebra over normalized data. Proc VLDB Endow 10(11):1214–1225. <https://doi.org/10.14778/3137628.3137633>
 36. Ghoting A, Krishnamurthy R, Pednault E, Reinwald B, Sindhwani V, Tatikonda S, Tian Y, Vaithyanathan S (2011) SystemML: Declarative machine learning on MapReduce. In: 2011 IEEE 27th International conference on data engineering, pp 231–242. <https://doi.org/10.1109/ICDE.2011.5767930>
 37. Li S, Chen L, Kumar A (2019) Enabling and optimizing non-linear feature interactions in factorized linear algebra. In: Proceedings of the 2019 International conference on management of data. SIGMOD '19, pp 1571–1588. ACM. <https://doi.org/10.1145/3299869.3319878>
 38. Abo Khamis M, Ngo HQ, Nguyen X, Olteanu D, Schleich M (2018) In-database learning with sparse tensors. In: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. SIGMOD/PODS '18, pp 325–340. ACM. <https://doi.org/10.1145/3196959.3196960>
 39. Richardson M, Domingos P (2006) Markov logic networks. Mach Learn 62(1):107–136. <https://doi.org/10.1007/s10994-006-5833-1>
 40. Getoor L (2013) Probabilistic soft logic: A scalable approach for markov random fields over continuous-valued variables. In: Proceedings of the 7th International conference on theory, practice, and applications of rules on the Web. RuleML'13, pp 1–1. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-39617-5_1
 41. Niu F, Ré C, Doan A, Shavlik J (2011) Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. Proc VLDB Endow 4(6):373–384. <https://doi.org/10.14778/1978665.1978669>
 42. Niu F, Zhang C, Re C, Shavlik J (2012) Scaling inference for markov logic via dual decomposition. In: Proceedings of the 2012 IEEE 12th International conference on data mining. ICDM '12, pp 1032–1037. IEEE Computer Society. <https://doi.org/10.1109/ICDM.2012.96>
 43. Zhang C, Ré C (2013) Towards high-throughput gibbs sampling at scale: A study across storage managers. In: Proceedings of the 2013 ACM SIGMOD International conference on management of data. SIGMOD '13, pp 397–408. ACM, New York. <https://doi.org/10.1145/2463676.2463702>
 44. Zhang C, Ré C, Sadeghian A, Shan Z, Shin J, Wang F, Wu S (2014) Feature engineering for knowledge base construction. IEEE Data Eng Bull
 45. Lu Y, Chowdhery A, Kandula S (2016) Optasia: A relational platform for efficient large-scale video analytics. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. SoCC '16, pp 57–70. ACM, New York, NY, USA. <https://doi.org/10.1145/2987550.2987564>

46. Zhang H, Ananthanarayanan G, Bodik P, Philipose M, Bahl P, Freedman MJ (2017) Live video analytics at scale with approximation and delay-tolerance. In: Proceedings of the 14th USENIX conference on networked systems design and implementation. NSDI'17, pp 377–392. USENIX Association., <http://dl.acm.org/citation.cfm?id=3154630.3154661>. Accessed 13 Oct 2019
47. Watcharapichat P, Morales VL, Fernandez RC, Pietzuch P (2016) Ako: Decentralised deep learning with partial gradient exchange. In: Proceedings of the Seventh ACM symposium on cloud computing. SoCC '16, pp 84–97. ACM. <https://doi.org/10.1145/2987550.2987586>
48. Duan S, Babu S (2007) Processing forecasting queries. In: Proceedings of the 33rd international conference on very large data bases. VLDB '07, pp 711–722. VLDB Endowment. <http://dl.acm.org/citation.cfm?id=1325851.1325933>. Accessed 13 Oct 2019
49. Fischer U (2015) Forecasting in database systems. In: Seidl, T, Ritter, N, Schöning, H, Sattler, K-U, Härder, T, Friedrich, S, Wingerath, W (eds.) Datenbanksysteme Für Business, Technologie und Web (BTW 2015), pp 483–492. Gesellschaft für Informatik e.V.
50. Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein J (2010) Graphlab: A new framework for parallel machine learning. In: UAI
51. Baumann P, Dehmel A, Furtado P, Ritsch R, Widmann N (1998) The multidimensional database system RasDaMan. In: Proceedings of the 1998 ACM SIGMOD International conference on management of data. SIGMOD '98, pp 575–577. ACM. <https://doi.org/10.1145/276304.276386>
52. Stonebraker M, Brown P, Poliakov A, Raman S (2011) The architecture of sciDB. In: Proceedings of the 23rd international conference on scientific and statistical database management. SSDBM'11, pp 1–16. Springer. <http://dl.acm.org/citation.cfm?id=2032397.2032399>. Accessed 13 Oct 2019
53. Huang B, Babu S, Yang J (2013) Cumulon: Optimizing statistical data analysis in the cloud. In: Proceedings of the 2013 ACM SIGMOD International conference on management of data. SIGMOD '13, pp 1–12. ACM, New York, NY, USA. <https://doi.org/10.1145/2463676.2465273>
54. Sparks ER, Talwalkar A, Haas D, Franklin MJ, Jordan MI, Kraska T (2015) Automating model search for large scale machine learning. In: Proceedings of the Sixth ACM symposium on cloud computing. SoCC '15, pp 368–380. ACM. <https://doi.org/10.1145/2806777.2806945>
55. Alexandrov A, Katsifodimos A, Krastev G, Markl V (2016) Implicit parallelism through deep language embedding. SIGMOD Rec 45(1):51–58. <https://doi.org/10.1145/2949741.2949754>
56. Russ R (2007) NetCDF-4 : Software implementing an enhanced data model for the geosciences
57. Baumann P (2016) Array Databases. In: Liu L, Özsu M (eds) Encyclopedia of Database Systems. Springer, New York, NY. https://doi.org/10.1007/978-1-4899-7993-3_2061-2
58. Baumann P, Mišev D, Merticariu V, Huu BP (2021) Array databases: concepts, standards, implementations. J Big Data 8:1–61. <https://doi.org/10.1186/s40537-020-00399-2>
59. Baumann P (1994) Management of multidimensional discrete data. VLDB J 3(4):401–444. <https://doi.org/10.1007/BF01231603>
60. Sarawagi S, Stonebraker M (1994) Efficient organization of large multidimensional arrays. In: Proceedings of 1994 IEEE 10th International conference on data engineering, pp 328–336. <https://doi.org/10.1109/ICDE.1994.283048>
61. Liaukevich V, Mišev D, Baumann P, Merticariu V (2017) Location and processing aware datacube caching. In: Proceedings of the 29th international conference on scientific and statistical database management. SSDBM '17, pp 34–1346. ACM. <https://doi.org/10.1145/3085504.3085539>
62. Cheng Y, Rusu F (2013) Astronomical data processing in EXTASCID. In: Proceedings of the 25th international conference on scientific and statistical database management. SSDBM, pp. 47–1474. ACM. <https://doi.org/10.1145/2484838.2484875>
63. Zhang Y, Kersten M, Ivanova M, Nes N (2011) SciQL: Bridging the gap between science and relational DBMS. In: Proceedings of the 15th Symposium on International Database Engineering & Applications. IDEAS '11, pp 124–133. ACM. <https://doi.org/10.1145/2076623.2076639>
64. PostGIS (2019) Post GIS Raster Manual. http://postgis.net/docs/manual-dev/using_raster_dataman.html. Accessed 14 Oct 2019
65. Teradata (2019) Array Data Type. <https://docs.teradata.com/r/Teradata-Database-SQL-Data-Types-and-Literals/June-2017/ARRAY/VARRAY-Data-Type>. Accessed 14 Oct 2019
66. GeoServer, Oracle Georaster User Manual (2019). <https://docs.geoserver.org/latest/en/user/data/raster/oraclegeoraster.html>. Accessed 14 Oct 2019
67. Information technology database languages — SQL — Part 15: Multi-dimensional arrays (SQL/MDA) (2019) Standard, International Organization for Standardization
68. Baumann P (2010) The OGC web coverage processing service (WCPS) standard. GeoInformatica 14(4):447–479. <https://doi.org/10.1007/s10707-009-0087-2>. Accessed 14 Oct 2019
69. Nexedi (2016) Wendelin.core Tutorial. <https://www.nexedi.com/wendelin-Core.Tutorial.2016>. Accessed 14 Oct 2019
70. TensorFlow (2019) An end-to-end open source machine learning platform. <https://www.tensorflow.org/>. Accessed 15 Oct 2019
71. Xtensor (2019) Multi-dimensional arrays with broadcasting and lazy computing. <https://xtensor.readthedocs.io/en/latest/>. Accessed 15 Oct 2019
72. OPeNDAP (2019) Advanced Software for Remote Data Retrieval. <https://www.opendap.org/>. Accessed 15 Oct 2019
73. Ophidia (2019) High Performance Data Mining & Analytics for eScience. <http://ophidia.cmcc.it/>. Accessed 15 Oct 2019
74. Google Earth Engine (2019) A planetary-scale platform for Earth science data & analysis. <https://earthengine.google.com/>. Accessed 15 Oct 2019
75. Papadopoulos S, Datta K, Madden S, Mattson T (2016) The TileDB array data storage manager. Proc VLDB Endow 10(4):349–360. <https://doi.org/10.14778/3025111.3025117>
76. Boost (2019) C++ Libraries. https://www.boost.org/doc/libs/1_71_0/libs/geometry/doc/html/index.html. Accessed 15 Oct 2019
77. Open Data Cube (2019) An Open Source Geospatial Data Management & Analysis Platform. <https://www.opendatacube.org/>. Accessed 15 Oct 2019
78. xarray (2019) N-D labeled arrays and datasets in Python. <http://xarray.pydata.org/en/stable/>. Accessed 15 Oct 2019
79. McKinney W (2010) Data structures for statistical computing in Python. In: Stéfan van der Walt, Jarrod Millman (eds.) Proceedings of the 9th python in science conference, pp 56–61. <https://doi.org/10.25080/Majora-92bf1922-00a>
80. Baumann P (2010) The OGC web coverage processing service (WCPS) standard. GeoInformatica 14(4):447–479. <https://doi.org/10.1007/s10707-009-0087-2>

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Dr. Sebastián Villarroya is assistant profesor at Universidade de Santiago de Compostela postdoctoral. He worked as research associate at Jacobs University Bremen and Universidade de Santiago de Compostela. He obtained his PhD at Universidade de Santiago de Compostela, researching on the integrated modeling and analysis of big raster and vector data. Beyond distributed big data analysis, sensor data acquisition systems and big spatial data analytics, he is

focused on the integration of machine learning technologies and raster database management systems.



Prof. Peter Baumann leads the L-SIS research group at Jacobs University Bremen. He pioneered the research field of Array Databases and is Principal Architect of the worldwide first complete and operational Array DBMS, rasdaman. He authored and co-authored 160+ book chapters and papers on array databases and related fields, and holds international patents. He received a series of national and international innovation awards. He is actively shaping Big Data

standards, such as the Open Geospatial Consortium (OGC) Big Geo Data suite and ISO SQL/MDA. He chairs several Big Data relevant groups. More info at: www.faculty.jacobs-university.de/pbaumann.