In-Database Text Classification with BornSQL

Emanuele Guidotti
University of Lugano
Lugano, Switzerland
emanuele.guidotti@usi.ch

Darya Shlyk
University of Milan
Milan, Italy
darya.shlyk@unimi.it

Stefano Montanelli
University of Milan
Milan, Italy
stefano.montanelli@unimi.it

Alfio Ferrara
University of Milan
Milan, Italy
alfio.ferrara@unimi.it

Abstract

The integration of databases and machine learning promises to enhance various aspects of data management, analysis, and application. However, in-database machine learning (In-DB ML) is not easily portable to different database management systems and current approaches are typically limited to training and inference, while modern machine learning pipelines often involve aspects such as continuous learning, unlearning, and explainability.

This paper presents BornSQL, a In-DB ML algorithm based on the Born Classifier [7], and exclusively implemented through standard SQL queries. BornSQL can handle categorical data, and it is particularly appropriate for classification of textual data. Further contributions of BornSQL are i) incremental learning to efficiently enforce model updates when new data become available in the db, ii) unlearning when selected data needs to be excluded due to privacy issues, and iii) global/local explainability to associate the importance of a feature/attribute in determining the classification result.

We illustrate the usage and scalability of the algorithm using a benchmark database consisting of 2,359,828 scientific publications divided into three classes and composed of 3,942,559 features. The training time is linear in the number of publications and the average inference time for a publication is 1 millisecond on our experimental environment. We discuss potential applications such as cost-effective model serving, exploratory data analysis, and data privacy.

Keywords

In-database machine learning, text classification

1 Introduction

In-Database Machine Learning (In-DB ML) is the practice of making Machine Learning (ML) pipelines executable from inside Relational Database Management Systems (RDBMS). There are several key arguments supporting the adoption of In-DB ML. In particular, bringing computations close to the data reduces data transfer, enforces query integration, and addresses security-related issues on sensitive data. Moreover, In-DB ML can benefit from data and model scalability, declarative programming interface, and efficient processing with optimized execution planning [12, 18].

Major database vendors, including Oracle, IBM, Microsoft, and Google, have recognized these benefits and are increasingly incorporating In-DB ML functionalities into their products. Existing solutions are designed as packages to use inside a DBMS, and characterized by SQL-based implementations of state-of-the-art ML algorithms.

A common solution is to provide User-Defined Functions (UDFs) in SQL to be integrated in the RDBMS for supporting ML operations like model training and prediction. For instance, this is the approach followed by Apache MADlib that relies on PostgreSQL's UDFs and additional Python drivers to enforce ML iterations according to either supervised and unsupervised methods [8]. A limitation of this kind of solutions is the security risk due to possible vulnerable code in UDFs. A further limitation is the portability of UDFs, since a solution developed (and maybe optimized) for a RDBMS by relying on a specific SQL dialect cannot be fully compatible with other platforms without code adaptations that can be complex. Furthermore, the portability of a solution from one implementation to another cannot be ensured in all the possible cases, limiting its applicability to specific RDBMS products (e.g., GaussML [13]).

An alternative to UDFs is the use of "hybrid" approaches like MASQ [3], where trained models and ML pipelines implemented in scikit-learn are compiled in standard SQL statements for compatibility with any RDBMS. Although data movements from/to the database are still limited, part of the computation is performed outside the DBMS with consequent latencies and scalability challenges.

Databases (dbs) are able to store any kind of data format in structured/tabular form, spanning from textual to numerical ones with several extensions and customization with respect to standard SQL. Textual data types, like character, character varying, and CLOB (Character Large OBject), are massively used and textual values/fields represent rich, informative datasources stored inside dbs. Surprisingly, none of the above families of In-DB solutions are currently supporting a ML approach/algorithm specifically focused on text analysis and classification. We argue that this is probably due to the need of dedicated libraries for text processing that are required for text preparation in many ML algorithms. Such a need represents an additional challenge for solutions based on UDFs or hybrid approaches, since it can further reduce code portability, by also shifting the computation outside the DBMS with consequent security, latency, and scalability issues.

As a final remark, we note that ML pipelines have grown in complexity. Besides training and inference, they include learning incrementally as new data become available, unlearning specific training data to meet privacy regulations, and explaining the reasons behind predictions. All this facets are not fully supported by current In-DB ML implementations.

This paper presents **BornSQL**, that is, to the best of our knowledge, the first text classification algorithm implemented in standard SQL that can learn, unlearn, make predictions, and explain.

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.19

BornSQL is an implementation of the Born Classifier originally introduced in [7] that is exclusively based on SQL queries. As such, BornSQL can be easily implemented in any RDBMS; it is highly portable, and immediately benefits from the query optimizer, data privacy and security, and scalability of the DBMS.

BornSQL is suitable for databases with categorical attributes or where continuous values can be discretized in a natural way. As such, BornSQL is particularly appropriate for classification of textual data. In BornSQL, attributes are treated as words in a text, and a database entity is considered as a superposition of attributes like text can be regarded as a superposition of words. For each entity, BornSQL generates its feature vector by computing the probability distribution over its attributes directly from the normalized structure of the database. Operations on the feature vectors are performed via SQL queries that operate on tables representing sparse tensors, and common table expressions are employed to avoid materializing intermediate results.

For the sake of usage, BornSQL is released as a Python package generating the SQL queries and implementing all routines in a specified RDBMS, as illustrated in this paper. [1]

This paper is organized as follows. Section 2 summarizes the Born Classifier proposed in [7] and constructs an exact incremental learning and unlearning process. Section 3 describes our implementation that exploits SQL tables to represent sparse tensors and SQL queries to perform mathematical operations upon them. Section 4 illustrates the usage of BornSQL and its scalability on a benchmark database of scientific publications. Section 5 discusses the comparison against MADlib, a potential competitor of our BornSQL. Section 6 reviews related works. Section 7 discusses potential applications. Finally, Section 8 gives our concluding remarks.

2 The Born Classifier

Let $\mathbf{x}$ be a feature vector with elements $x_j \geq 0$ for $j = 1, 2, \ldots$ that represents an unnormalized probability distribution over the features. Let $\mathbf{y}$ be a target vector with elements $y_k \geq 0$ for $k = 1, 2, \ldots$ that represents an unnormalized probability distribution over the classes. The goal is to predict the probability distribution over the classes from the feature vector and select the class with maximum probability for classification.

2.1 Training

Let $\mathcal{D} = \{(\mathbf{x}, \mathbf{y}, w)_n : n = 1, 2, \ldots\}$ be a training set containing, for each element $n = 1, 2, \ldots$, the corresponding feature vector $\mathbf{x}_n$, target vector $\mathbf{y}_n$, and a sample weight $w_n \geq 0$. The Born Classifier $\mathcal{B}$ trained on the dataset $\mathcal{D}$ produces the parameters $P = \mathcal{B}(\mathcal{D})$ with elements:

$$P_{jk} = \sum_n \frac{w_n x_{nj} y_{nk}}{\sum_{jk} x_{nj} y_{nk}} = \sum_{\mathcal{D}} w \frac{\mathbf{x} \otimes \mathbf{y}}{|\ \mathbf{x}\ ||\ \mathbf{y}\ |}, \tag{1}$$

which represent the unnormalized joint probability of features and classes, with the addition of arbitrary sample weights.

2.1.1 Incremental learning. As new data become available, the parameters can be updated with an exact incremental learning process.

Definition 2.1 (Exact incremental learning). Given a learning algorithm $\mathcal{A}(\cdot)$, a dataset $\mathcal{D}$, and a new dataset $\mathcal{D}_i$, we say the process $\mathcal{L}$ is an exact incremental learning process for $\mathcal{A}$ if and only if:

$$\mathcal{L}(\mathcal{A}(\mathcal{D}), \mathcal{D}_i) = \mathcal{A}(\mathcal{D} \cup \mathcal{D}_i)\,. \tag{2}$$

In other words, an exact incremental learning process ensures that the model is the same whether trained on the full dataset at once or in batches. An exact incremental learning process for $\mathcal{B}$ is:

$$\mathcal{L}(\mathcal{B}(\mathcal{D}), \mathcal{D}_i) = \mathcal{B}(\mathcal{D}) + \mathcal{B}(\mathcal{D}_i)\,. \tag{3}$$

Indeed:

$$\mathcal{B}(\mathcal{D}) + \mathcal{B}(\mathcal{D}_i) = \sum_{\mathcal{D} \cup \mathcal{D}_i} w \frac{\mathbf{x} \otimes \mathbf{y}}{|\ \mathbf{x}\ ||\ \mathbf{y}\ |} = \mathcal{B}(\mathcal{D} \cup \mathcal{D}_i)\,. \tag{4}$$

2.1.2 Unlearning. As training data are required to be deleted, the parameters can be updated using an exact unlearning process.

Definition 2.2 (Exact unlearning). Given a learning algorithm $\mathcal{A}(\cdot)$, a dataset $\mathcal{D}$, and a forget set $\mathcal{D}_f \subseteq \mathcal{D}$, we say the process $\mathcal{U}$ is an exact unlearning process for $\mathcal{A}$ if and only if:

$$\mathcal{U}(\mathcal{A}(\mathcal{D}), \mathcal{D}_f) = \mathcal{A}(\mathcal{D} \setminus \mathcal{D}_f)\,. \tag{5}$$

In other words, an exact unlearning process ensures that the unlearned model is equivalent to a model re-trained on the remaining data. Note that our definition is a special case of, e.g., [29] because the learning algorithm $\mathcal{A}$ is not randomized, and the unlearning process $\mathcal{U}$ is allowed to depend on $\mathcal{D}$ only through $\mathcal{A}$. An exact unlearning process for $\mathcal{B}$ is:

$$\mathcal{U}(\mathcal{B}(\mathcal{D}), \mathcal{D}_f) = \mathcal{L}(\mathcal{B}(\mathcal{D}), -\mathcal{D}_f)\,. \tag{6}$$

where $\mathcal{L}$ is given in (3) and $-\mathcal{D}_f = \{(\mathbf{x}, \mathbf{y}, -w) : (\mathbf{x}, \mathbf{y}, w) \in \mathcal{D}_f\}$. Indeed:

$$\mathcal{L}(\mathcal{B}(\mathcal{D}), -\mathcal{D}_f) = \sum_{\mathcal{D} \setminus \mathcal{D}_f} w \frac{\mathbf{x} \otimes \mathbf{y}}{|\ \mathbf{x}\ ||\ \mathbf{y}\ |} = \mathcal{B}(\mathcal{D} \setminus \mathcal{D}_f)\,. \tag{7}$$

2.2 Inference

The Born Classifier predicts the probability distribution over the classes from the feature vector $\mathbf{x}$ of a test item and the hyper-parameters $a > 0$, $b \geq 0$, and $h \geq 0$. First, it normalizes the joint probability P_{jk}:

$$W_{jk} = \frac{P_{jk}}{(\sum_j P_{jk})^b (\sum_k P_{jk})^{1-b}}\,. \tag{8}$$

Then, it computes the conditional probability of the k-th class given the j-th feature:

$$\mathcal{H}_{jk} = \frac{W_{jk}}{\sum_k W_{jk}}\,. \tag{9}$$

Next, it calculates the entropy for each feature and scales it so that it ranges between zero and one:

$$H_j = 1 + \frac{\sum_k \mathcal{H}_{jk} \ln(\mathcal{H}_{jk})}{\ln(\sum_k 1)}\,. \tag{10}$$

Finally, it computes the unnormalized probability of the k-th class:

$$u_k = \left(\sum_j H_j^h W_{jk}^a x_j^a\right)^{\frac{1}{a}}, \tag{11}$$

and the normalized probability $u_k / \sum_k u_k$, or directly selects the class $k^* = \text{argmax}_k\, u_k^a$ for classification.

2.2.1 Hyper-parameters. Hyper-parameter tuning can be performed without retraining the model because the training phase does not depend on the model's hyper-parameters. Moreover, for a given choice of (a, b, h), the weights $H_j^h W_{jk}^a$ can be precomputed and cached to speed up inference time.

[1] The package is available at https://github.com/eguidotti/bornrule.

2.3 Explainability

The contribution of the j-th feature to the probability u_k (local explanation) is given by the addend $H_j^h W_{jk}^a x_j^a$ in (11), while the contribution at the class level (global explanation) is obtained from the product $H_j^h W_{jk}^a$ regardless of the feature vector $\mathbf{x}$. For more details, the reader is referred to [7].

3 The Born Classifier in SQL

Here, we present our SQL implementation of the Born Classifier described in Section 2, namely BornSQL. Starting from normalized data in a relational database, we discuss how to transform the data, fit the classifier, deploy the model, make predictions, and explain them. An illustration is given in Figure 1.

Our implementation design consists of representing mathematical objects with Common Table Expressions (CTEs) and performing mathematical operations with standard SQL. Throughout the paper, we use the following notation. A tensor $T_{njk\ldots}$ is implemented as a table `T_njk...` with columns (`n`, `j`, `k`, ..., `w`) where the columns `n`, `j`, `k`, ... are the indices of the tensor and `w` is the corresponding value. We use the index n for the data items, j for the features, and k for the classes.

3.1 Preprocessing

The feature vectors $\mathbf{x}_n$ for $n = 1, 2, \ldots$ are represented with a table `X_nj`. We let the user provide a custom query q_x that transforms the original data Ω into such format.

$$\texttt{X\_nj}: \quad q_x(\Omega) \tag{12}$$

```
01 | SELECT n, j, w FROM ...
```

Similarly, the target vectors $\mathbf{y}_n$ for $n = 1, 2, \ldots$ are represented with a table `Y_nk` that is specified arbitrarily by the user with a query q_y.

$$\texttt{Y\_nk}: \quad q_y(\Omega) \tag{13}$$

```
01 | SELECT n, k, w FROM ...
```

Optionally, the user may also provide a query q_w to construct a table `W_n` giving the sample weights w_n for $n = 1, 2, \ldots$. Otherwise, the default $w_n = 1$ is used.

$$\texttt{W\_n}: \quad q_w(\Omega) \tag{14}$$

```
01 | SELECT n, w FROM ...
```

Finally, the user specifies which items to use in the previous queries by providing a query q_n that returns a table `N_n` containing the identifiers of the selected items:

$$\texttt{N\_n}: \quad q_n(\Omega) \tag{15}$$

```
01 | SELECT n FROM ... WHERE ...
```

Tables `X_nj`, `Y_nk` and `W_n` are filtered to include only the items in `N_n`. To improve the computational efficiency in case the query q_x uses `UNION` commands to combine the result set of two or more `SELECT` statements, our implementation allows the user to pass each `SELECT` statement individually so that they are filtered before combining their result sets.

3.2 Training

We train the classifier by computing the weights P_{jk} in (1) with the following queries.

$$\texttt{XY\_njk}: \quad x_{nj} y_{nk} \tag{16}$$

```
01 | SELECT
02 |     X_nj.n AS n,
03 |     X_nj.j AS j,
04 |     Y_nk.k AS k,
05 |     X_nj.w * Y_nk.w AS w
06 | FROM
07 |     X_nj, Y_nk
08 | WHERE
09 |     X_nj.n = Y_nk.n
```

$$\texttt{XY\_n}: \quad \textstyle\sum_{jk} x_{nj} y_{nk} \tag{17}$$

```
01 | SELECT n, SUM(w) AS w FROM XY_njk GROUP BY n
```

$$\texttt{P\_jk}: \quad P_{jk} \text{ in (1)} \tag{18}$$

```
01 | SELECT
02 |     XY_njk.j AS j,
03 |     XY_njk.k AS k,
04 |     SUM(W_n.w * XY_njk.w / XY_n.w) AS w
05 | FROM
06 |     XY_njk, XY_n, W_n
07 | WHERE
08 |     XY_njk.n = XY_n.n AND
09 |     XY_njk.n = W_n.n
10 | GROUP BY
11 |     XY_njk.j,
12 |     XY_njk.k
```

Finally, the weights P_{jk} are stored in the table `{model}_corpus`, where `{model}` is a custom prefix used to identify the model. This identifier is introduced to allow the creation of multiple models on the same database. The incremental learning in (3) is implemented by incrementing the weights in the corpus:

```
01 | INSERT INTO {model}_corpus (j, k, w)
02 | SELECT j, k, w FROM P_jk
03 | ON CONFLICT (j, k)
04 | DO UPDATE SET w = {model}_corpus.w + excluded.w
```

and the unlearning in (6) is implemented by using $-w_n$ instead of the original w_n.

3.3 Deployment

To deploy a model, we retrieve its hyper-parameters from the table `params`. This table contains the columns (`model`, `a`, `b`, `h`) where `model` is the primary key identifying the model, and the remaining columns are the corresponding hyper-parameters a, b, h.

$$\texttt{ABH}: \quad a, b, h \tag{19}$$

```
01 | SELECT a, b, h FROM params WHERE model = '{model}
   |     '
```

Then, the following queries compute the weights $H_j^h W_{jk}^a$ in (11) from the weights P_{jk} that are stored in the table `P_jk = {model}_corpus`.

$$\texttt{P\_j}: \quad \textstyle\sum_k P_{jk} \tag{20}$$

```
01 | SELECT j, SUM(w) AS w FROM P_jk GROUP BY j
```

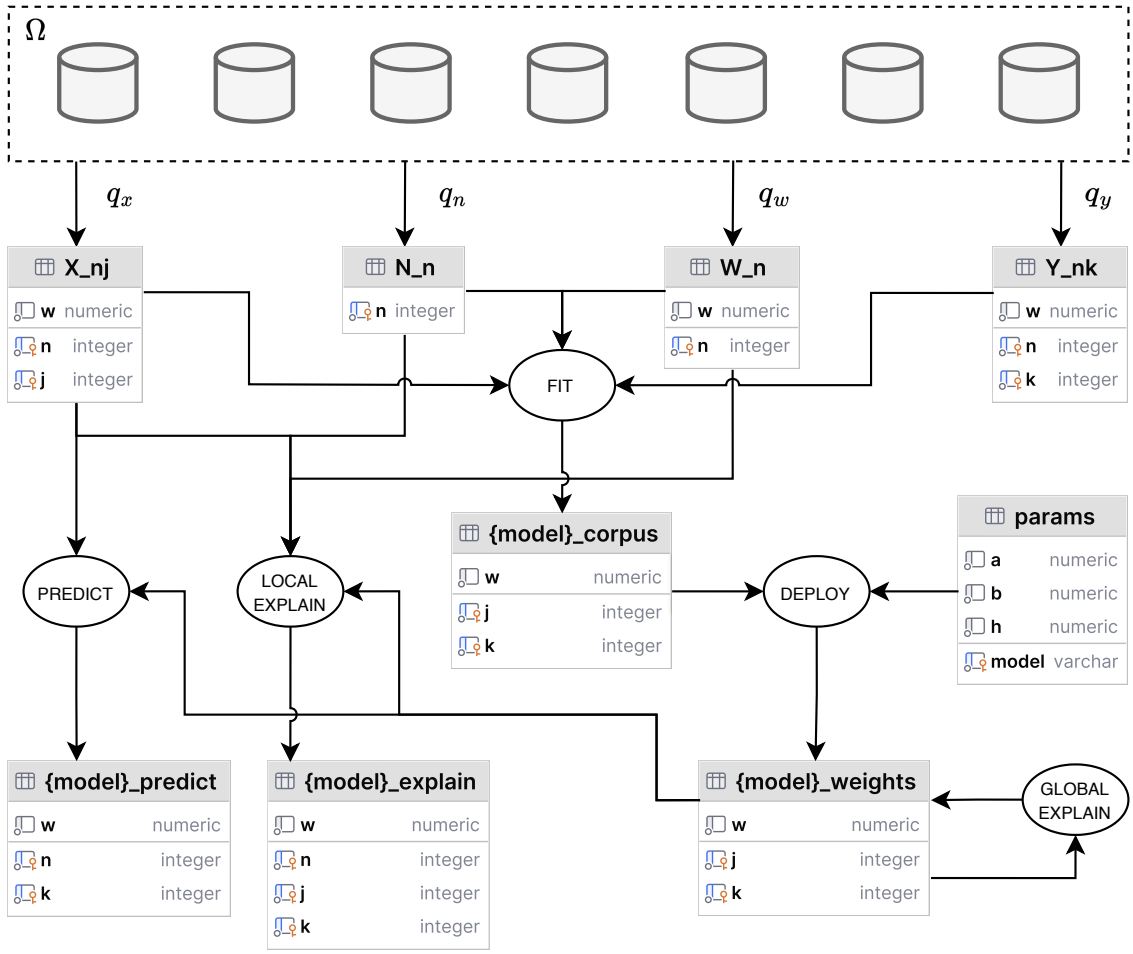


Figure 1: Implementation design of BornSQL.

P_k : $\sum_j P_{jk}$ (21)

```
SELECT k, SUM(w) AS w FROM P_jk GROUP BY k
```

W_jk : W_{jk} in (8) (22)

```
SELECT
    P_jk.j AS j,
    P_jk.k AS k,
    P_jk.w * POW(P_k.w, -b) * POW(P_j.w, b-1) AS
     w
FROM
    P_jk, P_j, P_k, ABH
WHERE
    P_jk.j = P_j.j AND
    P_jk.k = P_k.k
```

W_j : $\sum_k W_{jk}$ (23)

```
SELECT j, SUM(w) AS w FROM W_jk GROUP BY j
```

H_jk : $\mathcal{H}_{jk}$ in (9) (24)

```
SELECT
    W_jk.j AS j,
    W_jk.k AS k,
    W_jk.w / W_j.w AS w
FROM
    W_jk, W_j
WHERE
    W_jk.j = W_j.j
```

H_j : H_j in (10) (25)

```
SELECT
    H_jk.j AS j,
    1 + SUM(H_jk.w * LOG(H_jk.w)) / (
        SELECT LOG(COUNT(*)) FROM P_k
    ) AS w
FROM
    H_jk
GROUP BY
    H_jk.j
```

HW_jk : $H_j^h W_{jk}^a$ (26)

```
SELECT
    W_jk.j AS j,
    W_jk.k AS k,
    POW(H_j.w, h) * POW(W_jk.w, a) AS w
FROM
    W_jk, H_j, ABH
```

```
07 | WHERE
08 |     W_jk.j = H_j.j
```

Finally, the table `HW_jk` can be materialized and stored as the table `{model}_weights` to speed up inference times.

3.4 Inference

We compute u_k^a in (11) for each item:

$$\texttt{HWX\_nk}: \quad \sum_j H_j^h W_{jk}^a x_{nj}^a \tag{27}$$

```
01 | SELECT
02 |     X_nj.n AS n,
03 |     HW_jk.k AS k,
04 |     SUM(HW_jk.w * POW(X_nj.w, a)) AS w
05 | FROM
06 |     HW_jk, X_nj, ABH
07 | WHERE
08 |     HW_jk.j = X_nj.j
09 | GROUP BY
10 |     X_nj.n,
11 |     HW_jk.k
```

and select the class $k^* = \text{argmax}_k\, u_k^a$ for classification.

```
01 | SELECT
02 |     R_nk.n,
03 |     R_nk.k
04 | FROM (
05 |     SELECT
06 |         n,
07 |         k,
08 |         ROW_NUMBER() OVER(
09 |             PARTITION BY n ORDER BY w DESC
10 |         ) AS w
11 |     FROM
12 |         HWX_nk
13 |     ) AS R_nk
14 | WHERE
15 |     R_nk.w = 1
```

If classification probabilities are requested, we compute the unnormalized probabilities:

$$\texttt{U\_nk}: \quad \left(\sum_j H_j^h W_{jk}^a x_{nj}^a\right)^{\frac{1}{a}} \tag{28}$$

```
01 | SELECT n, k, POW(w, 1/a) AS w FROM HWX_nk, ABH
```

and their normalization factors:

$$\texttt{U\_n}: \quad \sum_k U_{nk} \tag{29}$$

```
01 | SELECT n, SUM(w) AS w FROM U_nk GROUP BY n
```

and return the normalized probabilities $u_{nk}/\sum_k u_{nk} \in [0, 1]$.

```
01 | SELECT
02 |     U_nk.n AS n,
03 |     U_nk.k AS k,
04 |     U_nk.w / U_n.w AS w
05 | FROM
06 |     U_nk, U_n
07 | WHERE
08 |     U_nk.n = U_n.n
```

3.5 Explainability

The global weights $H_j^h W_{jk}^a$ are obtained by reading the table `HW_jk`. These weights provide the feature importance for each class unconditionally from any test item.

```
01 | SELECT j, k, w FROM HW_jk
```

The local weights $H_j^h W_{jk}^a x_{nj}^a$ for the n-th test item provide the feature importance for each class conditional on that item. For multiple items, we first compute their (weighted) average feature vector:

$$\mathbf{z} = \sum_n w_n \frac{\mathbf{x}_n}{|\,\mathbf{x}_n\,|} \tag{30}$$

and then provide the local weights $H_j^h W_{jk}^a z_j^a$ for that vector. Such weights provide the feature importance for each class conditional on the average test item. We start by computing the normalization factor for each item.

$$\texttt{X\_n}: \quad \sum_j x_{nj} \tag{31}$$

```
01 | SELECT
02 |     X_nj.n AS n,
03 |     SUM(X_nj.w) AS w
04 | FROM
05 |     X_nj
06 | GROUP BY
07 |     X_nj.n
```

Then, we calculate the average vector $\mathbf{z}$ in (30).

$$\texttt{Z\_j}: \quad \sum_n \frac{w_n x_{nj}}{\sum_j x_{nj}} \tag{32}$$

```
01 | SELECT
02 |     X_nj.j,
03 |     SUM(W_n.w * X_nj.w / X_n.w) AS w
04 | FROM
05 |     X_nj, X_n, W_n
06 | WHERE
07 |     X_nj.n = X_n.n AND
08 |     X_nj.n = W_n.n
09 | GROUP BY
10 |     X_nj.j
```

Finally, we return the local weights $H_j^h W_{jk}^a z_j^a$.

```
01 | SELECT
02 |     HW_jk.j,
03 |     HW_jk.k,
04 |     HW_jk.w * POW(Z_j.w, a) AS w
05 | FROM
06 |     HW_jk, Z_j, ABH
07 | WHERE
08 |     HW_jk.j = Z_j.j
```

4 Experimental Evaluation

Here, we illustrate the usage of BornSQL and evaluate its scalability using a benchmark dataset of scientific publications. All results are obtained using Python 3.11.5 and PostgreSQL 12, MySQL 9.4.0, and SQLite 3.50.2 on a Ubuntu 20.04.6 LTS VM server with 32GB RAM and 20 assigned CPU cores. To demonstrate that BornSQL can also operate under constrained resources, we verified that all experiments can be replicated on a MacBook Air with an M2 chip and 8 GB RAM.

4.1 Data

The database used in the experiments contains metadata about scientific publications sourced from the Elsevier Scopus database. [2] It covers citations from 1990 to 2022 in three major subject categories in the field of data science, including *Artificial Intelligence*, *Statistics and Probability*, and *Decision Sciences*. In the database, the subject categories are represented with a 4-digit code (ASJC) assigned by the Scopus classification system, where the first two digits denote the macro subject area and the last two refer to related sub-fields. The distribution of macro subject areas is detailed in Table 1 below.

Table 1: Distribution of subject areas.

ASJC	Subject area	Count
1702	Artificial Intelligence	1,024,703
2613	Statistics and Probability	426,341
18XX	Decision Sciences	908,784
	Total:	2,359,828

The version of the database used in the experiments is constructed as follows. We start with a set of four tables extracted from the Scopus database containing information about publications, corresponding keywords, authors, and abstracts. First, we remove unnecessary fields as well as duplicate rows in all tables. Then, we order publications chronologically based on their publication date and create a sequential identifier, replacing the original Scopus IDs. Finally, we vectorize the abstract of each publication and cast all numerical fields to the integer data type.

The final database is stored as a relational database with three tables. The database schema is depicted in Figure 2. The table publication is the fact table with 2,359,828 rows and four columns: unique publication identifier (id), name of the publication venue (pubname), ASJC code assigned by Scopus (asjc), and the vectorized abstract (abstract). The tables *pub_author* (7,048,668 rows) and *pub_keyword* (8,301,637 rows) are dimension tables storing one-to-many associations of publications with corresponding author identifiers and keywords, respectively.

Figure 2: Database schema.

4.2 Preprocessing

The classification task consists of predicting the first two digits of the asjc code from the attributes pubname, authid, keyword, and abstract. Here we exemplify the queries q_x in (12), q_y in (13), and q_w in (14) that we use to set up this task.

First, we notice that the attributes pubname, authid, and keyword are categorical, and we use a one-hot encoding scheme for them. Specifically, we give unitary weight to each attribute associated with the items using the following queries.

[2] https://www.elsevier.com/products/scopus.

```
SELECT
    id as n,
    'pubname:'||pubname as j,
    1.0 as w
FROM publication
```

```
SELECT
    pubid as n,
    'authid:'||authid as j,
    1.0 as w
FROM pub_author
```

```
SELECT
    pubid as n,
    'keyword:'||keyword as j,
    1.0 as w
FROM pub_keyword
```

Then, we vectorize the abstract by counting how many times each lexeme appears in the text. The following query uses the `tsvector` data type, which is efficient but specific to PostgreSQL. For MySQL and SQLite we treat the abstract as `text` and vectorize it with similar queries using the functions `json_table` and `json_each`, respectively.

```
SELECT
    id as n,
    'abstract:'||lexeme as j,
    array_length(positions, 1) as w
FROM publication, unnest(abstract)
```

Finally, the query q_x is the `UNION ALL` of the previous queries, and it concatenates all the result sets. Notice that we prepend each feature with a prefix to avoid collisions between different attributes with the same value. An illustration is provided in Table 2.

Table 2: Example of a transformed item.

n	j	w
13	pubname:communications in statistics - ...	1.0
13	authid:7004218793	1.0
13	keyword:sampling efficiency	1.0
13	keyword:renewal	2.0
13	abstract:binomial	1.0
13	abstract:sample	7.0
13	abstract:sampling	7.0
13	abstract:variance	3.0
13	...	...

The query q_y selects the target category by selecting the first two digits of the asjc code, and each category is given a unitary weight. In our case, each item is associated with exactly one category. However, the same query would also be valid when an item is associated with multiple categories, and we want to give the same weight to all categories.

```
SELECT
    id as n,
    asjc / 100 AS k,
    1.0 AS w
FROM publication
```

Finally, to use unitary sample weights, the query q_w selects a weight equal to 1 for all items in the database. Our implementation is optimized to skip this step and uses unitary weights by default.

```
01 | SELECT id as n, 1.0 AS w FROM publication
```

4.3 Training

To train the model, we specify the query q_n in (15) that selects the identifiers of the training items from the database. This query is used to filter the queries q_x, q_y, and q_w and build the tables `X_nj`, `Y_nk` and `W_n` that are used to train the model as described in Section 3.2. For instance, the model can be trained on the first 100 items using the following query for q_n, or it can be trained on the entire database by omitting the `WHERE` clause.

```
01 | SELECT id as n FROM publication WHERE id <= 100
```

Figure 3 reports the training time as a function of the items used for training. To maintain a stationary distribution of the data across subsamples, we proceed as follows. The first subsample uses 10% of items by selecting the first every 10th item with the query:

```
01 | SELECT id as n FROM publication WHERE id % 10 <=
          0
```

Similarly, the second subsample uses 20% of items by selecting the first two for every 10th item with the query:

```
01 | SELECT id as n FROM publication WHERE id % 10 <=
          1
```

We continue by incrementing the subsamples until all items are considered. Figure 3 shows that, while the training time (fit) can vary depending on the specific DBMS, it is linear in the number of items, suggesting that model training is scalable to large databases.

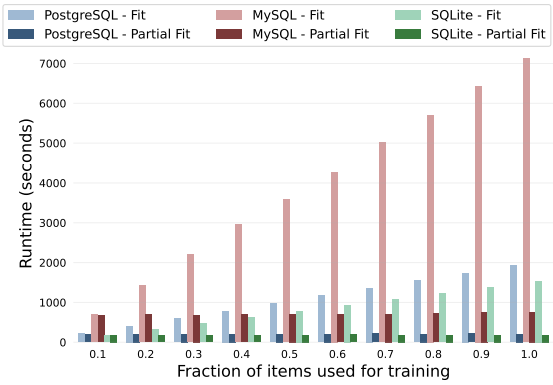


Figure 3: Training time.

4.3.1 Incremental learning. Instead of fitting from scratch every subsample, we can use the exact incremental learning process in (3) to update the model using only the new items. Figure 3 reports the incremental leaning (partial fit) runtime as new items are added to the training set. For all DBMSs, the runtime is approximately constant for equally-sized batches of new items, suggesting that the learning algorithm is suitable for dynamic settings where efficient model updates are needed as new data becomes available.

4.3.2 Unlearning. The incremental fit can also be employed to unlearn selected training items using negative sample weights as in (6). In this case, it is sufficient to multiply by -1 the weights in the query q_w in (14) that was used for training. For example, to unlearn items previously employed for training with default unitary weights, it is sufficient to specify the following query q_w and select the items to unlearn with q_n.

```
01 | SELECT id AS n, -1.0 AS w FROM publication
```

4.4 Deployment

After training, the model can be deployed to accelerate inference time in production. The deployment involves computing the weights discussed in Section 3.3 and storing them in a table, which can be readily used to multiply the feature vectors of the test items during inference. The deployment operation does not depend on the number of training items. Instead, it depends on the number of classes and features.

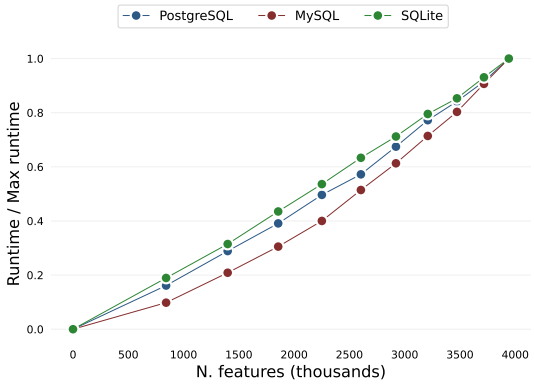
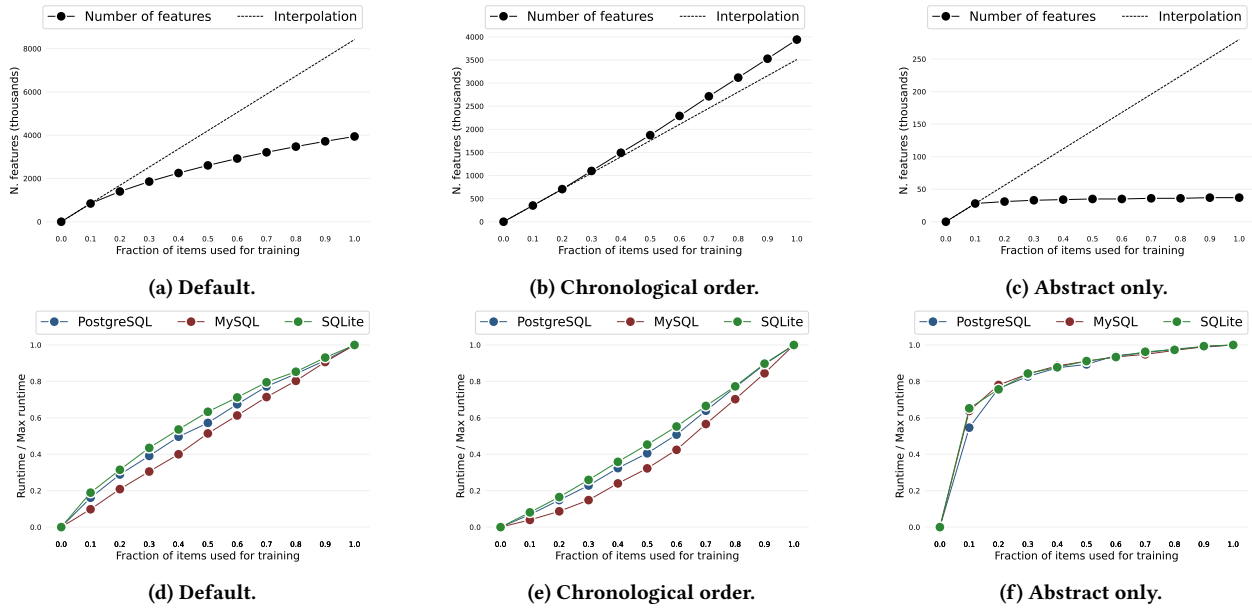


Figure 4: Deployment time and number of features.

As the number of classes is fixed to three, Figure 4 reports the normalized deployment time as a function of the number of features. The time to deploy the model trained on the entire database, consisting of 3,942,559 features, is 123s in PostgreSQL, 580s in MySQL, and 109s in SQLite. The deployment time is approximately linear in the number of features in agreement with the complexity analysis in [7]. The same scalability applies to the classes as the roles of classes and features are interchangeable.

While the deployment time does not directly depend on the number of training items, the number of features may depend on the number of items seen during training. Thus, the deployment time may indirectly depend on the number of training items. To investigate such dependence, we report three examples that proxy for three scenarios.

First, we split publications into batches as described in Section 4.3 to maintain a stationary distribution of the data across different subsamples. This scenario proxies for the case when the distribution of the data is stationary, and thus, it is expected that the probability of observing a new feature decreases as the training set increases. We count the number of features seen during training as the fraction of training items increases from 0% to 100% and measure the corresponding deployment times. The



(a) Default. (b) Chronological order. (c) Abstract only.

(d) Default. (e) Chronological order. (f) Abstract only.

Figure 5: Number of features (a-b-c) and normalized deployment time (d-e-f) for the three scenarios described in Section 4.4. The dotted line is the interpolation line passing through the first two points.

results are displayed in Figure 5, Panels (a) and (d), respectively. The increase in the number of features is sublinear in the number of training items, corresponding to a (sub)linear increase in the deployment time.

Second, we split publications in chronological order. Here, the first batch of data corresponds to the oldest publications, and the last batch contains the most recent ones. This scenario proxies for the case where the distribution of the data is non-stationary, and the total number of features is potentially unbounded. We count the number of features seen during training as the fraction of training items increases from 0% to 100% and measure the corresponding deployment times. The results are displayed in Figure 5, Panels (b) and (e), respectively. We find that the number of features grows (super)linearly with the number of training items. Indeed, most recent publications are typically associated with a larger number of authors, more keywords, and longer abstracts, bringing in an increasing number of new features per item. In this case, the deployment time also grows (super)linearly.

Third, we split publications to maintain a stationary distribution of the data as in the first case. However, we only use the abstract to construct the feature vector this time. This scenario proxies cases where the total number of features is finite, and most features are seen with a small training set. The number of features and the corresponding deployment times are displayed in Figure 5, Panels (c) and (d), respectively. In this scenario, the number of features and the deployment times stabilize quickly, and they are essentially constant regardless of the number of training items.

Overall, we conclude that unless new training items bring in an ever-increasing number of features, the deployment time is (sub)linear in the number of items and scalable to very large datasets, especially when the number of possible features is finite.

4.5 Inference

To predict a test instance, we specify the query q_n in (15) that selects its identifier. This query filters the query q_x and builds the table `X_nj` that is used for inference as described in Section 3.4. For example, we use the following query q_n to predict the publication number 13.

```
01 | SELECT 13 as n
```

We analyze the time needed to classify a single item depending on the model size. First, we split publications into batches to maintain a stationary distribution of the data as described in Section 3.2. Then, we fit the model with an amount of training data ranging from 10% to 100% of the total dataset size. Finally, we predict the test publication number 13 and report the corresponding inference times in Figure 6. Using 10% of data, the inference time is 7 seconds, and it grows to 38 seconds when the model is trained on the entire database. We notice the following. On the one hand, the inference time is less than the deployment time in Section 4.4, showing that the query optimizer is correctly optimizing the query by computing only the weights needed for the classification of the test instance. On the other hand, by pre-computing all weights, the prediction step would essentially reduce to a `JOIN` of the weights with the input data and a `GROUP BY` operation, and the inference time may reduce dramatically, as we analyze next.

4.5.1 Deployment. Instead of predicting a test instance by computing the weights on the fly each time, we can first deploy the model by pre-computing all weights described in Section 3.3 and then perform inference with such weights. Figure 6 shows that, after deployment, the inference time drops significantly, below one second, across all DBMSs. To better assess the actual inference time per single item, we predict the first 1000 items with the following query.

```
01 | SELECT id as n FROM publication WHERE id <= 1000
```

The total inference time is 0.91s in PostgreSQL, 1.02s in MySQL, and 0.62s in SQLite, corresponding to an average inference time of about one millisecond per item.

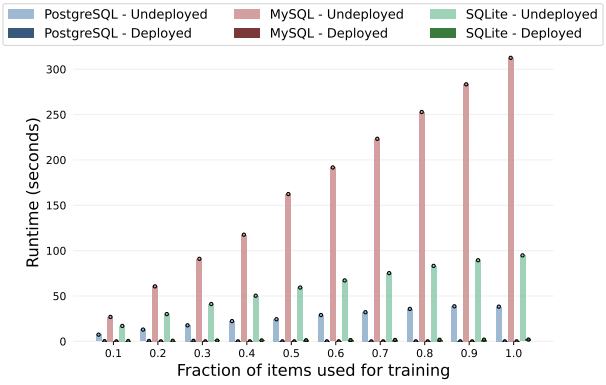


Figure 6: Inference time for a single item.

4.6 Explainability

In this section, we illustrate the global and local explanations discussed in Section 3.5.

4.6.1 Global explanation. The global explanation provides the feature importance for each class unconditional from any test item. In Table 3, we report the first three features with the highest weights for each of the three classes *Artificial Intelligence* ($k = 17$), *Statistics and Probability* ($k = 26$), and *Decision Sciences* ($k = 18$). As a general remark, we note that the publication venue is the most important feature to predict the subject area. Not surprisingly, the journals in Table 3 represent publication venues that are frequently occurring in the respective classes within our dataset. We also find that the word "robot" used in the abstract indicates a publication in the field of *Artificial Intelligence*.

Table 3: Global explanation.

k	j	w
17	abstract:robot	0.033
17	pubname:studies in computational intelligence	0.027
17	pubname:physics of life reviews	0.020
17	...	...
18	pubname:quality progress	0.040
18	pubname:international series in operations research and management science	0.033
18	pubname:zwf zeitschrift fuer wirtschaftlichen fabrikbetrieb	0.028
18	...	...
26	pubname:physical review - statistical, nonlinear, and soft matter physics	0.040
26	pubname:statistics in medicine	0.038
26	pubname:journal of mathematical sciences	0.037
26	...	...

4.6.2 Local explanation. The local explanation provides the feature importance for each class conditional on a test item. In Table 4, we report the top ten features with the highest weight for the example publication number 13. The predicted and actual class is *Statistics and Probability* ($k = 26$), and, according to the first row of Table 4, the first reason for this prediction is that the article is published in "Communications in Statistics - Theory and Methods". To further support this prediction, the terms "random", "sample", and "variance" in the abstract have a larger weight for *Statistics and Probability* ($k = 26$) than for *Decision Sciences* ($k = 18$) and *Artificial Intelligence* ($k = 17$).

Table 4: Local explanation.

k	j	w
26	pubname:communications in statistics - theory and methods	0.0024
26	abstract:random	0.0019
26	abstract:sample	0.0015
26	abstract:variance	0.0014
18	abstract:random	0.0012
18	abstract:sample	0.0010
18	abstract:variance	0.0009
26	abstract:poisson	0.0009
17	abstract:sample	0.0008
17	abstract:random	0.0008
...	...	...

5 Comparison with MADlib

Here we compare BornSQL with popular classification algorithms implemented in MADlib [8], namely logistic regression, supported vector machines, and decision trees, using PostgreSQL. We will comment on the impossibility of using large datasets with high-dimensional features with MADlib, such as the Scopus dataset illustrated in the previous section, and we will use two complementary datasets: Adult [1] and RLCP [25]. The task of the Adult dataset is to predict whether annual income of an individual exceeds $50K/yr based on census data. There are a total of 11,687 positive and 37,155 negative examples. The dataset is split into 32,561 instances belonging to the training set and 16,281 instances belonging to the test set. We use the categorical variables in the census data, which consist of 102 one-hot encoded features. The task of the RLCP dataset is to decide from comparison patterns whether the underlying records belong to one person. We convert all columns into match or non-match and one-hot encode them, for a total of 18 features. The dataset contains 5,749,132 instances (20,931 positive and 5,728,201 negative). We use the first 4,600,000 instances for training and the remaining ones for testing.

5.1 Data handling

One of the main differences between our implementation of BornSQL and other algorithms implemented by MADlib is the way the data are handled.

BornSQL acts directly on the normalized structure of the database, without requiring the materialization of any intermediate result, and natively supports sparse computations in standard SQL with the queries introduced in Section 3.

On the contrary, MADlib requires to prepare the input data, and materialize them, in one of the following three formats. The first format is a classical tidy-data format [28] where rows are observations and each column is a feature. However, the DBMS typically limits the maximum number of columns in a table and thus this format cannot be used when the number of features is above this limit. To overcome this limitation, the second format stores all the features in a single column that contains an array of values with arbitrary fixed length. However, all the elements in

each array must be stored explicitly and this is inefficient when most of the elements are zero, as it is typically the case with text data. The last option is to use a sparse format that only stores the non-zero elements. Unfortunately, this format is only provided for storage purposes and the current version of MADlib (2.1.0) does not support training algorithms or performing inference with sparse input. This is an important limitation, as it essentially prevents from using high-dimensional categorical data with any algorithm. For instance, consider the Scopus dataset in the previous section with approximately 2 million rows and 4 million features. Assuming 4 bytes for an integer, storing these data in a non-sparse format requires $2 \times 4 \times 4 = 32$ terabytes, which is thousands times larger than the original database size, and prevented us from training any algorithm in MADlib. We thus report below a comparison on the Adult and RLCP datasets, which have a small set of features that MADlib can handle efficiently.

5.2 Runtimes

On the Adult and RLCP datasets, the training and inference times of BornSQL are of the same order of magnitude of the classifiers implemented in MADlib. On the adult dataset, BornSQL requires 3.25s to train, 0.01s to deploy, and 0.86s to predict all the test instances. The data preprocessing step in MADlib requires 1.45s and the training time for Decision Trees (DT), Supported Vector Machines (SVM), and Logistic Regression (LR) are 57.73s, 25.89s, and 19.69s, respectively. The corresponding inference times are 0.73s, 0.27s, and 0.29s. On the RLCP dataset, BornSQL requires 184.10s to train, 0.01s to deploy, and 58.05s for inference. The data preprocessing step in MADlib requires 52.88s, and the training (inference) times are 306.67s (21.41s) for DT, 1,425.45s (10.29s) for SVM, and 265.22s (5.63s) for LR.

5.3 Evaluation metrics

We execute all the algorithms with default hyper-parameters and report the evaluation metrics on the test set in Table 5. Overall, BornSQL is comparable with the popular algorithms implemented in MADlib. Specifically, it achieves lower precision and higher recall than the other algorithms on the Adult dataset, which is expected, as the Born Classifier natively normalizes by the class imbalance [7]. In the RCLP dataset, BornSQL matches the precision of the other classifiers, while also achieving higher recall.

The supplementary code also adds the 20 Newsgroup (20NG) and Reuters (R8 and R52) datasets to the benchmark. BornSQL achieves an accuracy of 87.3% on 20NG, 95.4% on R8, and 88.0% on R52, which fully replicate the original results in [7]. As the classification performance is independent from our SQL implementation, benchmarking against other state-of-the-art classifiers is outside the scope of our evaluation and we refer the reader to [7] for additional comparative tests in this regard.

5.4 Explainability

It is worth noting that BornSQL natively provides explanations alongside its predictions, a feature that is completely missing in MADlib. For instance, by inspecting the global explanation on the Adult dataset, we find that the features "native_country: Outlying-US(Guam-USVI-etc)" and "native_country: Holand-Netherlands" have positive weight for the negative class but zero weight for the positive class. This means that these nationalities have never been seen in the positive class and suggests that the data may not contain a representative sample for these nationalities, creating potential biases in the predictions. By querying the database, we confirm that the training data only contain 14 instances with "native_country" equal to "Outlying-US(Guam-USVI-etc)" and only 1 instance with "native_country" equal to "Holand-Netherlands". Overall, this example illustrates how BornSQL may be used to spot potential biases in the training data before feeding them to machine-learning algorithms that may otherwise start discriminating based on under-represented characteristics.

Table 5: Macro-averaged precision (Prc.), recall (Rec.), and F1-score for BornSQL, Decision Tree (DT), Supported Vector Machines (SVM), and Logistic Regression (LR) on the Adult and RLCP datasets.

	Adult			RLCP		
	Prc.	**Rec.**	**F1 Score**	**Prc.**	**Rec.**	**F1 Score**
BornSQL	0.70	**0.78**	0.70	**0.99**	**1.00**	**0.99**
DT	0.77	0.71	0.73	**0.99**	0.97	0.98
SVM	**0.78**	0.72	0.74	**0.99**	0.97	0.98
LR	**0.78**	0.73	**0.75**	**0.99**	0.97	0.98

6 Related Work

To our knowledge, BornSQL is the first classifier for in-database machine learning that supports learning, unlearning, inference, and explainability with standard SQL. Since no direct benchmarks exist that support all these features simultaneously, we review related works in several dimensions relevant to its capabilities.

MADlib. MADlib [8] provides a comprehensive framework for in-database analytics. Developed for PostgreSQL and Greenplum databases, it offers a high-level interface for training and deploying machine learning models. The library features an extensive collection of ML algorithms designed to harness optimized in-database processing capabilities. However, unlike BornSQL, ML pipelines in MADlib cannot directly run on the normalized data typical of relational databases. For training ML models using MADlib, the data must be denormalized and materialized into an input table, introducing several inefficiencies due to the materialization of the join results and extensive data manipulation required to convert the data into formats suitable for ML processing. A significant limitation is the inability to use sparse data for training and inference, which makes it impossible to process large datasets with high-dimensional features.

Overall, the main differences between MADlib and BornSQL are that i) BornSQL is a classification algorithm particularly suitable for high-dimensional categorical features while MADlib offers an extensive choice of ML algorithms for both classification and regression tasks particularly suitable for low-dimensional categorical and numerical features, ii) MADlib is an SQL wrapper for routines implemented in C++ for PostgreSQL and Greenplum databases while BornSQL relies exclusively on standard SQL queries and it is portable to any relational database, and iii) BornSQL offers additional routines for incremental learning, unlearning, and explainability.

In-DB ML. Machine learning over relational data can be cast as a database problem [18]. Considerable work on enabling machine learning workflow inside database systems adopt SQL-based approaches that reuse existing database capabilities and often extend the DBMS with custom data structures and query

operators [2, 5, 14, 16]. Despite being less efficient than the SQL-only solutions, approaches based on User Defined Functions (UDFs) provide a viable alternative, where a more familiar procedural programming framework can be used to embed existing machine learning algorithms into SQL code [4, 6, 8]. Sandha et al. [24] realize distributed learning using parallel model training and implement C-based ML primitives for model training and validation inside the Teradata SQL engine. Following a line of work on large-scale statistical computation in RDBMS, Jankov et al. [10] apply targeted changes to SimSQL, a distributed RDBMS on Hadoop MapReduce, to allow for concise declarative specification and efficient execution of complex recursive computations. Building on their work, Schüle et al. [26] deploy machine learning pipelines in standard SQL, integrating gradient descent as a native operator for efficient automatic differentiation inside the Umbra database engine and accelerating in-database model training by off-loading the processing to GPU kernels. Steffen Kläbe et al. [12] evaluate several approaches for in-database inference on deep neural models. Overall, the above works present approaches to implement machine learning models into DBMSs using UDFs, C-APIs, or SQL translations for simple models. This work adopts the latter approach and implements the classification algorithm in [7] using standard SQL without any external runtime, in a similar way to JoinBoost [9], which rewrites tree training algorithms over normalized databases into pure SQL.

In-DB Inference. MASQ [3, 19] provides an automatic framework for compiling scikit-learn [20] predictive pipelines into SQL queries that get executed in SQL-compliant engines. ML models are trained outside the DBMS and pushed into the database to run the inference step. InferDB [23] approximates end-to-end inference pipelines using a light-weight embedding to improves inference times while maintaining similar prediction accuracy compared to the pipeline it approximates. Compared to these solutions, BornSQL supports both the training and the inference steps in SQL.

In-DB XAI. Explainable Artificial Intelligence (XAI) studies the explainability and interpretability of machine learning models [27]. Several frameworks, such as LIME [21] and SHAP [15], produce model-agnostic explanations by approximating the decision process of black-box models. However, it is argued that trying to explain black-box models, rather than creating models that are interpretable in the first place, is likely to perpetuate bad practice [22]. In this context, BornSQL is not a black-box model that needs to be explained with methods such as LIME and SHAP. Indeed, it is inherently interpretable and directly yields both local and global explanations with perfect fidelity and faithfulness to the model's actual behavior, entirely within standard SQL.

In-DB Unlearning. Machine unlearning [11, 29, 30] refers to the process of removing the influence of specific training data from a trained model's weights, ensuring compliance with privacy regulations like the GDPR's "right to be forgotten" [17]. Unlearning in-database has the potential to enhance data privacy by allowing seamless data removal directly within the database, improving compliance with privacy regulations, and ensuring data consistency between the database and the model by synchronizing the data removal process with the model's weights. To our knowledge, the field of in-database unlearning is largely underexplored, and this work represents a first step in this direction.

7 Discussion

In this section, we outline potential application areas in which BornSQL could offer unique advantages and finally discuss its main limitations.

Cost-effective model serving. A fitted instance of BornSQL only requires storing a tuple of hyper-parameters, a table containing the model's weights, and an optional table to speed up inference times. The storage cost of the model is that of creating one or two tables with three columns and a number of rows equal to the number of unique pairs of features and classes. Moreover, only the table used for inference may be retained to reduce storage costs further if the model is not planned to be updated. Inference is implemented by querying the database. As such, the model may be served by implementing an Application Programming Interface (API) that runs SQL queries, leveraging the concurrency and scalability of the database without the need of any additional resource.

External data. In some application contexts, it may be necessary to train the model with data external to the database without incurring the cost of storing the training set. In such a situation, instead of specifying SQL queries that select and transform data stored in the database, training samples can be processed externally to compute the weights P_{jk} in Section 3.2 and update the table `{model}_corpus` accordingly, without the need to import the data into the database. We also mention that inference is not limited to items stored in the database. Indeed, the table `X_nj` used for inference does not necessarily need to be created by the queries q_n and q_x using data from the database, but it may also be constructed externally and written to a temporary table when needed.

Explainable predictions and exploratory data analysis. Alongside its predictions, BornSQL returns local and global explanations. Such explanations may be used to understand the reasons behind predictions and validate the model. They may also be used to spot potential biases in the training data before feeding them to other machine-learning pipelines. Generally, BornSQL may be used not only as a classifier but also as a tool for exploratory data analysis.

Continuous learning and privacy regulations. Some applications require continuous model updates as new training data become available. Other applications require models to unlearn subsets of training data to meet privacy regulations. BornSQL may be helpful in these applications as it can be trained with exact incremental learning and an exact unlearning process. For instance, when a user withdraws consent to data processing, a trigger may run the query to unlearn the corresponding data and ensure that the model is always consistent with the user's consent.

Limitations. Although BornSQL demonstrates robust capabilities in categorical data classification, its primary limitation lies in its inability to directly handle continuous attributes or regression tasks. Indeed, continuous variables should first undergo discretization, which might restrict the effective use of BornSQL primarily to databases with mostly categorical attributes. Consequently, the functionalities presented in this paper might not be universally applicable across all types of datasets.

8 Conclusion

This paper addresses the challenge of executing machine learning workflows entirely inside a relational database management system. We present BornSQL, an implementation of the Born classifier [7] that only requires standard SQL, without any external procedural code. The key capabilities of BornSQL include: (i) training a text classification model using only SQL queries; (ii) incremental learning, updating the model efficiently as new data arrives; (iii) unlearning, removing or "forgetting" specific training data upon request (e.g., for privacy compliance); and (iv) explainability, providing both global and local explanations. The paper's central idea is to treat relational data attributes (especially categorical or text attributes) as features analogous to words in a document, enabling native text classification within the DB. Overall, BornSQL is the first in-DB ML solution to support training, inference, incremental updates, unlearning, and explainability all within SQL, offering a portable and secure approach to ML on data within relational databases.

Artifacts

All code and data to reproduce our results are available at https://github.com/eguidotti/bornrule.

References

[1] Barry Becker and Ronny Kohavi. 1996. Adult. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20.

[2] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *12th Conference on Innovative Data Systems Research, CIDR 2022,Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, Chaminade, CA, USA, 6 pages.

[3] Francesco Del Buono, Matteo Paganelli, Paolo Sottovia, Matteo Interlandi, and Francesco Guerra. 2021. Transforming ML Predictive Pipelines into SQL with MASQ. In *Proc. of the Int. Conference on Management of Data SIGMOD2021*. Association for Computing Machinery, Virtual Event, China, 2696–2700.

[4] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA: Abstraction for advanced in-database analytics. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1400–1413.

[5] Len Du. 2020. In-Machine-Learning Database: Reimagining Deep Learning with Old-School SQL. *CoRR* abs/2004.05366 (2020).

[6] Christian Duta, Denis Hirn, and Torsten Grust. 2020. Compiling PL/SQL Away. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, Amsterdam, The Netherlands, 6 pages.

[7] Emanuele Guidotti and Alfio Ferrara. 2022. Text Classification with Born's Rule. In *Advances in Neural Information Processing Systems*, Vol. 35. Curran Associates, Inc., New Orleans, LA, USA, 30990–31001.

[8] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711.

[9] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees over Normalized Data Using Only SQL. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3071–3084.

[10] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. 2020. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *ACM SIGMOD Record* 49, 1 (2020), 43–50.

[11] Yiwen Jiang, Shenglong Liu, Tao Zhao, Wei Li, and Xianzhou Gao. 2022. Machine unlearning survey. In *Fifth International Conference on Mechatronics and Computer Technology Engineering (MCTE 2022)*, Dalin Zhang (Ed.), Vol. 12500. SPIE, Chongqing, China, 125006J.

[12] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. 2023. Exploration of Approaches for In-Database ML. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, Ioannina, Greece, 311–323.

[13] Guoliang Li, Ji Sun, Lijie Xu, Shifu Li, Jiang Wang, and Wen Nie. 2024. GaussML: An End-to-End In-Database Machine Learning System. In *Proc. of the IEEE 40th Int. Conference on Data Engineering (ICDE)*. Utrecht, Netherlands, 5198–5210.

[14] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1933–1936.

[15] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). Curran Associates, Inc., Long Beach, CA, USA, 4765–4774.

[16] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2018. Scalable linear algebra on a relational database system. *ACM SIGMOD Record* 47, 1 (2018), 24–31.

[17] Alessandro Mantelero. 2013. The EU Proposal for a General Data Protection Regulation and the roots of the 'right to be forgotten'. *Computer Law & Security Review* 29, 3 (2013), 229–235.

[18] Dan Olteanu. 2020. The relational data borg is learning. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3502–3515.

[19] Matteo Paganelli, Paolo Sottovia, Kwanghyun Park, Matteo Interlandi, and Francesco Guerra. 2023. Pushing ML Predictions Into DBMSs. *IEEE Transactions on Knowledge and Data Engineering* 35, 10 (2023), 10295–10308.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[21] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the Demonstrations Session, NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*. The Association for Computational Linguistics, San Diego, California, USA, 97–101.

[22] Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence* 1, 5 (2019), 206–215.

[23] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1830–1842.

[24] Sandeep Singh Sandha, Wellington Cabrera, Mohammed Al-Kateb, Sanjay Nair, and Mani B. Srivastava. 2019. In-database Distributed Machine Learning: Demonstration using Teradata SQL Engine. *Proc. VLDB Endow.* 12, 12 (2019), 1854–1857.

[25] Irene Schmidtmann, Gael Hammer, Murat Sariyar, and Aslihan Gerhold-Ay. 2009. Record Linkage Comparison Patterns. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C51K6B.

[26] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*, Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar (Eds.). ACM, Tampa, FL, USA, 25–36.

[27] Erico Tjoa and Cuntai Guan. 2020. A survey on explainable artificial intelligence (xai): Toward medical xai. *IEEE transactions on neural networks and learning systems* 32, 11 (2020), 4793–4813.

[28] Hadley Wickham. 2014. Tidy data. *Journal of statistical software* 59 (2014), 1–23.

[29] Heng Xu, Tianqing Zhu, Lefeng Zhang, Wanlei Zhou, and Philip S. Yu. 2023. Machine Unlearning: A Survey. *ACM Comput. Surv.* 56, 1 (aug 2023), 36 pages.

[30] Haibo Zhang, Toru Nakamura, Takamasa Isohara, and Kouichi Sakurai. 2023. A review on machine unlearning. *SN Computer Science* 4, 4 (2023), 337.