InferDB: In-Database Machine Learning Inference Using Indexes

Ricardo Salazar-Díaz
Hasso Plattner Institute
University of Potsdam
ricardo.salazardiaz@hpi.de

Boris Glavic
University of Illinois Chicago
bglavic@uic.edu

Tilmann Rabl
Hasso Plattner Institute
University of Potsdam
tilmann.rabl@hpi.de

ABSTRACT

The performance of inference with machine learning (ML) models and its integration with analytical query processing have become critical bottlenecks for data analysis in many organizations. An ML inference pipeline typically consists of a preprocessing workflow followed by prediction with an ML model. Current approaches for in-database inference implement preprocessing operators and ML algorithms in the database either natively, by transpiling code to SQL, or by executing user-defined functions in guest languages such as Python. In this work, we present a radically different approach that approximates an end-to-end inference pipeline (preprocessing plus prediction) using a light-weight embedding that discretizes a carefully selected subset of the input features and an index that maps data points in the embedding space to aggregated predictions of an ML model. We replace a complex preprocessing workflow and model-based inference with a simple feature transformation and an index lookup. Our framework improves inference latency by several orders of magnitude while maintaining similar prediction accuracy compared to the pipeline it approximates.

PVLDB Reference Format:
Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. InferDB: In-Database Machine Learning Inference Using Indexes. PVLDB, 17(8): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hpides/inferdb.

1 INTRODUCTION

Machine learning models are deployed in a wide variety of applications, e.g., customer segmentation, recommender systems, process automation [47]. While training a model can be expensive, a single model is often used to make many predictions. Thus, the performance of *inference* has become a critical bottleneck for these applications. For instance, cloud vendors report that inference workloads are responsible for 90% of all ML infrastructure costs [19]. As the data that is the input for inference is often managed using database systems, inference functionality needs to be tightly integrated into DBMS, e.g., Zhang et al. [53] report around 25% of the Greenplum customers use ML alongside SQL analytics.

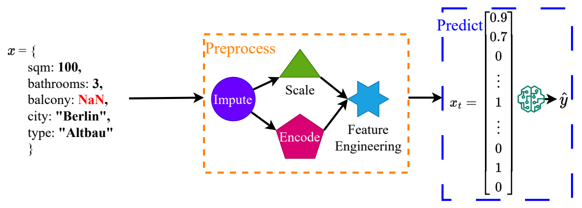
The need for preprocessing and inference inside DBMSs has led to significant interest in the database community. Prior work either (i) implements preprocessing and ML operators inside the DBMS or (ii) provides access to ML runtimes from within the database. Work that falls within the first category translates operators into SQL or user-defined functions [1, 26, 37, 46, 48, 50], or implements new operators within the database engine [4, 11, 53]. Recent work [39, 43] has shown that the number of operators per pipeline ranges from tens to more than a thousand, with many operators being custom transformations highly specific to particular tasks and datasets. Either the operators are treated as black boxes during query optimization, or the system's optimizer has to be extended to support the new operators. Approaches for (ii) integrate ML runtimes into the DBMS to access ML operators from within the database [28, 39, 45]. This reuses existing ML operator implementations but has to move data between the database and ML runtime. It also lacks integration with the query optimizer. In summary, existing approaches either result in large development efforts or require extensive data movement and are often not integrated well with the DBMS optimizer and execution engine.

In this work, we present *InferDB*, which approximates preprocessing steps (e.g., data transformation, missing value imputation) and model-based prediction using a lightweight embedding and a traditional index structure. We demonstrate that significant performance improvements for inference can be achieved when our approach is deployed within a DBMS or as a standalone application. By relying on existing infrastructure and extensibility mechanisms available in every DBMS, our approach enables the seamless integration of inference into SQL queries without requiring any changes to the database. InferDB is based on the assumption that the predictions for a data point can be feasibly approximated using predictions for similar data points. A straightforward way to realize this is to index the prediction for all training data points (before preprocessing) and, at inference time, approximate the prediction for a test data point x by aggregating the predictions for a sufficiently large set of nearest neighbors (k-NN) of x in the training data. This approach has several drawbacks as the size of the index is linear in the size of the training data, and while there is a lot of work optimizing k-NN search, it is not as performant as a simple index lookup and less integrated with most DBMS. InferDB learns a lightweight embedding that discretizes the input features and selects the most relevant discretized features to build the index on. This has the important advantage of having a single index entry for all training data points equal in the embedding space (reducing index size), and that inference is a point lookup.

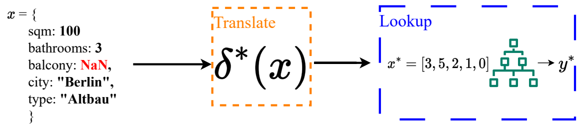
Example 1. *Figure 1 shows an example of predicting real estate prices for a test data point x with an inference pipeline (Figure 1a) and with InferDB (Figure 1b). In Figure 1a, the preprocessing workflow is applied to x to transform it into the numerical format. The preprocessing pipeline imputes missing values, scales numerical values,*

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Preprint



(a) Inference pipeline: preprocessing and prediction



(b) InferDB inference pipeline: translation and lookup

Figure 1: Inference pipeline (Figure 1a) and index-based inference (Figure 1b) for predicting property prices.

and applies one-hot encoding for categorical values. New features are constructed by combining the values of the original features. The transformed data point x_t is passed to a regressor to predict the price $\hat{y}$. In contrast, our approach (Figure 1b) maps x into an embedding space consisting of a carefully selected set of discretized features. A single lookup in the index storing aggregated predictions of the model returns a prediction for the discretized data point x^*.

Important factors for the accuracy and performance of InferDB are (i) the right choice of embedding space (binning) and (ii) the selection of discretized features to build the index on. As we are aggregating predictions of the model for all data points that share the same representation in the embedding space, it is critical for the accuracy of our approach that the predictions of the model are well preserved in the embedding space. Towards this goal, we rely on existing supervised discretization techniques (we use the OptBinning [34] framework). We present a greedy heuristic to select a subset $\mathcal{X}^*$ of the discretized features $\tilde{\mathcal{X}}$ by repeatedly selecting a feature that results in the largest improvement in prediction performance compared to the set selected so far.

Our approach is agnostic to the index structures that store the predictions for discretized training data points. An important advantage of using a standard index structure is that, in contrast to the alternatives discussed above, our approach synergizes well with the query optimizer and execution engine of the DBMS. This enables efficient evaluation of queries that combine prediction with relational operators. As we demonstrate in our experimental evaluation, replacing costly preprocessing steps and prediction using a model with a lightweight embedding and index lookup can significantly improve the performance of queries involving inference.

InferDB's current implementation focuses on classic in-DB ML tasks such as regression, binary classification, and multi-label classification on structured data, typically stored in DBMS relations. In our experiments, we report InferDB results for an image recognition task and discuss the challenges and ideas to support unstructured data workloads in the future. Our main contributions are:

- We present InferDB, a framework for approximating ML inference pipelines using index structures available in DBMS.
- We exploit discretization techniques that take a model's predictions into account to bin features such that predictions are mostly preserved in the embedding space and present a greedy heuristic for selecting binned features for indexing.
- We implement InferDB as a standalone index and in Postgres using standard database index structures.
- We experimentally compare InferDB against state-of-the-art standalone techniques and other frameworks for in-DBMS ML inference. InferDB achieves competitive accuracy and reduces inference latency by up to two orders of magnitude.

Outline: In Section 2, we introduce relevant background on ML training and inference. Section 3 provides an overview of InferDB. In Section 4, we discuss supervised discretization and describe how to build an index that approximates an end-to-end inference pipeline. Then, we explain how to perform inference using InferDB in Section 5. We discuss related work in Section 6, experimentally evaluate our framework in Section 7, and conclude in Section 8.

2 BACKGROUND

ML Training. Consider a training dataset $D_{train} = \{(x_i, y_i)\}$ over a set of features $\mathcal{X}$ and an outcome $\mathcal{Y}$ (e.g., $\mathcal{Y} = \{0, 1\}$ for binary classification), where a data point x_i is a vector of values (one for each feature in $\mathcal{X}$) and $y_i \in \mathcal{Y}$. In ML training, a preprocessing workflow $\mathcal{P}$ is a graph of preprocessing operators. We treat this workflow as a black box function that, given D_{train} as input, outputs a dataset $\mathcal{P}(D_{train})$ with features $\mathcal{X}^{pre}$ and outcome $\mathcal{Y}$. Typically, $\mathcal{P}$ will preserve the outcome for each training data point.

InferDB is agnostic to the operations applied by $\mathcal{P}$. Dataset $\mathcal{P}$ (D_{train}) is then the input to the learning phase, which trains an ML model $f : \mathcal{X}^{pre} \rightarrow \mathcal{Y}$. The model f maps preprocessed data points to predictions from $\mathcal{Y}$. We use $\hat{y}_i$ to denote the predicted outcome produced by f for data point x_i, i.e., $f(\mathcal{P}(x_i)) = \hat{y}_i$. We briefly review a few common preprocessing operators in the following.

Example 2. *Figure 3 shows a typical training pipeline. Most ML algorithms expect training data to be numerical. The training data must be preprocessed before training a model **(6)**. Missing values and outliers may be fixed using an imputer **(1)**. Imputers can be the mean value for numerical features and the most frequent value for categorical features, trained classifiers, or regressors to impute missing values [30]. Some data types require preprocessing to extract numerical features **(2)**. Distance-based and gradient-descent-based ML algorithms are sensitive to unscaled data, so numerical features are often scaled **(3)**, e.g., using min-max (normalization) and Z-score (standardization) [43]. Categorical features are transformed into a numerical representation using one-hot coding **(4)**. Feature engineering operators are applied during preprocessing **(5)** to increase the predictive power of the features [17, 43]. Feature construction combines the values of two or more features with linear and nonlinear operators to generate new features with higher predictive power [24].*

Inference Pipelines. To use a model f for inference on a test dataset $D_{test} = \{x_i\}$ with features $\mathcal{X}$, the test data passes through

preprint

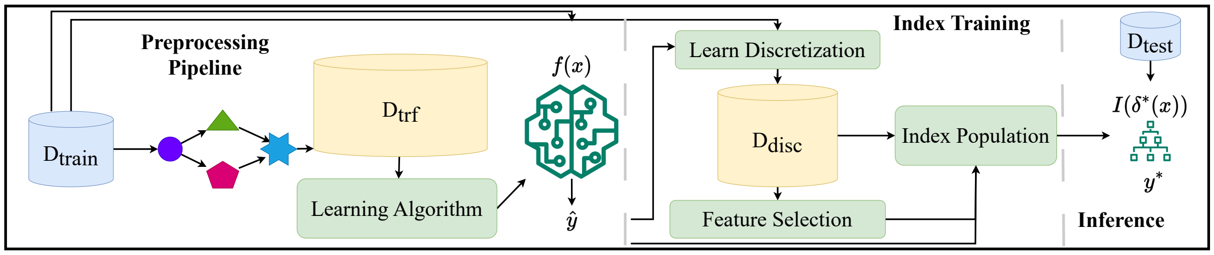


Figure 2: Overview of the InferDB framework: A regular training pipeline is used for model training. A subset of discretized features is selected, forming the keys for the populated index. At inference time, a data point x is mapped into the embedding space to retrieve its prediction $I(\delta^*(x))$ using the index / prediction table.

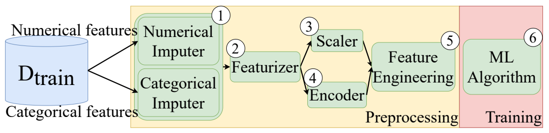


Figure 3: Example of a training pipeline comprising a preprocessing workflow and training.

the same preprocessing pipeline $\mathcal{P}$ used during training[1] to generate the input data $\mathcal{P}(D_{test})$ for prediction with the model. Afterwards, the model f is used to compute a prediction $\hat{y}$ for each $x \in D_{test}$:

$$\hat{y} = f(\mathcal{P}(x))$$

In this work, we are mainly concerned with improving the runtime of inference. We use $t_{preprocess}$ to denote the time for applying $\mathcal{P}$ to transform a test data point (or set of test data points) and $t_{predict}$ to denote the time for making a prediction using f for the transformed data point (set of points). Our goal is to minimize the end-to-end latency $t_{inference}$ for inference observed by the user:

$$t_{inference} = t_{preprocess} + t_{predict}$$

3 SYSTEM OVERVIEW

In this section, we provide an overview of our InferDB framework. Figure 2 shows the three main steps in building and deploying an index approximating an inference pipeline: training, index training, and inference. First, a training pipeline consisting of a preprocessing workflow and a training step generates a model f as discussed in Section 2.

Index Training & Population. Based on the output of the training pipeline, InferDB learns a discretized embedding δ and then selects a subset of the discretized features to build the index on. We learn the embedding δ by binning each input feature $X \in \mathcal{X}$ such that the predictions of the model f are preserved. We use $\tilde{\mathcal{X}}$ to denote the discretized features. Afterwards, we select a subset $\mathcal{X}^* \subseteq \tilde{\mathcal{X}}$ to build the index. We use $\delta^* : \mathcal{X} \times \mathcal{Y} \rightarrow \mathcal{X}^* \times \mathcal{Y}$ to denote the transformation that maps a data point in the original feature space $\mathcal{X}$ into the embedding space $\mathcal{X}^*$. This step aims to reduce the size of the index structure we will build over the training data and exclude less relevant features for prediction.

We then create an index $\mathcal{I}$ that maps each training data point $x \in D_{train}$ into the embedding space using δ^*. Each distinct data point x^* is then used as an index key and associated with an aggregated prediction generated from the predictions made by the model over all data points that are equal in the embedding space using an aggregation function α. We use majority vote for binary and multi-label classification and average for regression.

The index $\mathcal{I}$ produced by this process encodes an approximation of the predictions made by the inference pipeline; all data points that are equal in the embedding space receive the same prediction. $\mathcal{I}$ is either implemented as a standalone index structure, or we create a table in a DBMS (which we refer to as a *prediction table*) to store the predictions and create standard indexes on this table.

Index-Based Inference. During inference, the index replaces the end-to-end inference pipeline. Each test data point $x \in D_{test}$ is mapped into the embedding space, resulting in a data point $x^* = \delta^*(x)$. The prediction for x is retrieved using a lookup: $y^* = \mathcal{I}(x^*)$.

4 PREDICTION TABLE AND INDEX CREATION

In this section, we describe how to generate an indexed database table (*prediction table*) or standalone index that approximates an inference pipeline. The preprocessing pipeline $\mathcal{P}$, raw input data D_{train}, and model f trained over the output of $\mathcal{P}$ on D_{train} are given as input. To build the index, we first learn an embedding space by creating discretized features $\tilde{\mathcal{X}}$ from the features $\mathcal{X}$ of D_{train} such that the labels of training data points are preserved in the discretized space as much as possible (Section 4.1). Then, we select a subset $\mathcal{X}^*$ of the discretized features $\tilde{\mathcal{X}}$ as keys for the index. The goal of this step is to identify a small set of features (to reduce index size) that are relevant for prediction (Section 4.2). To build the prediction table and index structure (Section 4.3) we map the training data D_{train} into the embedding space $\mathcal{X}^*$ and aggregate the predictions for all training data points that are mapped to the same point x^* in $\mathcal{X}^*$ as the prediction for x^*. The overall goal is to build a prediction table of reasonable size that provides a robust approximation of the predictions made by the inference pipeline. In Section 4.4, we discuss using a standalone trie index instead of a DBMS. We use $t_{learn-embed}$ to denote the time for learning

[1]Some preprocessing operators require training an ML model or require the tuning of parameters. For our work, it is irrelevant how the preprocessing pipeline was produced, and we consider only the version of the preprocessing pipeline that was used to train the model we want to use for inference.

Preprint

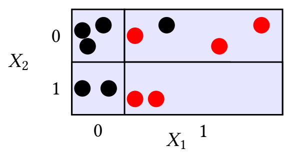


Figure 4: Example illustrating discretizing D_{train} with two features X_1 and X_2 for a binary classification problem.

the embedding $\tilde{\mathcal{X}}$ for all features in $\mathcal{X}$, $t_{feature-sel}$ for selecting a subset of features $\mathcal{X}^*$ to build the index on, and $t_{populate}$ for populating the index. Thus, $t_{build-index}$, the runtime overhead for the training pipeline incurred by our approach is:

$$t_{build-index} = t_{learn-embed} + t_{feature-sel} + t_{populate}$$

4.1 Discretization

The first step towards approximating inference using our framework is to learn an embedding of the input space $\mathcal{X}$ by discretizing each input feature $X \in \mathcal{X}$. We use $\tilde{\mathcal{X}}$ to denote the discretized features and $\tilde{X}$ to denote the discretized version of feature $X \in \mathcal{X}$. The domain of each feature (whether numerical or categorical) will be partitioned into a small number of bins. For this approximation to be accurate, the model's predictions must be preserved to the maximum degree possible in the embedding space.

Example 3 (Discretization). *Figure 4 shows an example discretization for a binary classification. The training data has two features $\mathcal{X} = \{X_1, X_2\}$. The class assigned by the model is shown as the color of the training data point. Both features have been split into two bins. The bins are mostly homogenous, i.e., training data points with the same label are grouped into bins, and there is little uncertainty about the predictions for a bin.*

Discretization Techniques. Discretization has been studied extensively by the ML community [9, 10, 31]. As our goal is to preserve the predictions of the model f, we employ a supervised discretization technique, which groups values considering the mutual information between the bins and the predictions of the model f (called the target feature in this context). For each numerical feature $X_i \in \mathcal{X}$ with domain $\mathbb{D}_i$ we want to generate a set of $\#B_i$ bins with lower and upper bounds $B_i = \{b_{ij} = [l_{ij}, u_{ij}]\}$ such that B_i is a partitioning of the domain of X_i, i.e., the bins are pairwise disjoint and the union of these ranges covers the domain of X_i. For any categorical feature X_i with domain $\mathbb{D} = \{c_i\}$, each bin is a subset of $\mathbb{D}$, and again, the bins are disjoint, and their union covers $\mathbb{D}$. $\delta : \mathcal{X} \times \mathcal{Y} \rightarrow \tilde{\mathcal{X}} \times \mathcal{Y}$ denotes the function that takes a data point in the original feature space $\mathcal{X}$ and maps this data point into the discretized space $\tilde{\mathcal{X}}$ and $D_{disc} = \delta(D_{train})$ denotes the discretized version of the training dataset. $D[X]$ and $x[X]$ denote the projection of datasets/data points onto a feature or set of features.

Information Value: Measuring Discretization Quality. *Jeffrey's divergence* [20], also known as *information value* (*IV*), is widely employed [34] as a measure for the quality of a discretization, i.e., the uncertainty about the model's prediction within each bin. In the following, consider a single feature X and the discrete version $\tilde{X}$ of this feature with bins B produced by a discretization technique.

For binary classification ($\mathcal{Y} = \{0, 1\}$), let P be the probability distribution of the Class 1 over $D_{disc}[\tilde{X}, \mathcal{Y}]$, i.e., over the discretized training dataset projected on $\tilde{X}$ and the prediction of the model f. Let D_i denote the subset of the training data points whose value in feature X falls within bin b_i:

$$D_i = \{x \mid x \in D_{train} \land \delta(x)[\tilde{X}] \in b_i\}$$

For a bin $b_i \in B$, let p_i denote the probability of predicting Class 1 for training data points in b_i. That is, p_i is the fraction of training data points whose value in X is in b_i and which receive the prediction of 1 by the model (after preprocessing):

$$p_i = \frac{\sum_{x_j \in D_i} \mathbb{1}[\hat{y}_j = 1]}{N_i} \quad \textbf{for} \quad N_i = |D_i|$$

We use Q (and q_i) to denote the analog concepts for Class 0 (as $p_i + q_i = 1$, we have $q_i = 1 - p_i$). Following a one-vs-all approach, we can compute p_i and q_i for the multi-label classification case. The formulas for computing the information value for binary and multi-label classification tasks of a discretized feature $\tilde{X}$ with bins B and labels L (for the multi-label case) are shown below. The main goal of discretization is to choose bins so that $\text{IV}(\tilde{X})$ is maximized, maximizing the expected difference between the probabilities p_i and q_i. This is equivalent to minimizing the uncertainty about the model's prediction per bin, as one class will have a high probability of being predicted for data points in the bin, and the probability of the other class is low within the bin.

$$\text{IV}(\tilde{X}) = \sum_{i=1}^{|B|} (p_i - q_i) \log \frac{p_i}{q_i} \qquad \text{(binary classification)}$$

$$\text{IV}(\tilde{X}) = \sum_{k=1}^{|L|} \sum_{i=1}^{|B|} (p_{ik} - q_{ik}) \log \frac{p_{ik}}{q_{ik}} \qquad \text{(multi-label classification)}$$

Equation (1) shows the formula for computing IV for the case where $\mathcal{Y}$ is continuous, i.e., regression tasks. A good binning solution for regression maximizes the divergence between the local mean μ_i of bin b_i and the global mean μ, let $N = |D_{train}|$.

$$\text{IV}(\tilde{X}) = \sum_{i=1}^{|B|} |\mu - \mu_i| \frac{N_i}{N}$$
$$\textbf{for} \quad \mu = \frac{\sum_{x_j \in D_{train}} \hat{y}_j}{N} \quad \textbf{and} \quad \mu_i = \frac{\sum_{x_j \in D_i} \hat{y}_j}{N_i} \tag{1}$$

OptBinning. We use the open-source Python package *OptBinning* [34] for discretization as it optimizes for $\text{IV}(\tilde{\mathcal{X}})$, is reasonably efficient, supports both categorical (classification) and continuous (regression) target features, and supports additional constraints on the solution such as a maximum number of bins per feature.

4.2 Feature Selection

The discretized feature space $\tilde{\mathcal{X}}$ has the same number of features as the original feature space $\mathcal{X}$. As models are often built over 10s of features or more, building a prediction table over $\tilde{\mathcal{X}}$ directly would result in a sparsely filled index where for most data points $\tilde{x}$ in D_{disc} there will be no training data point or only a small number of training data points that are equal to $\tilde{x}$ when mapped into the embedding space. This means insufficient information to approximate

Preprint

Algorithm 1 Greedy Feature Selection

```
Input: discretized training dataset $D_{disc}$, discretized features $\tilde{\mathcal{X}}$, bins $\{B_i\}$
Output: Sorted list $\bar{\mathcal{X}}^*$ of selected discretized features.
1: $L_{\tilde{\mathcal{X}}} \leftarrow$ sortDesc($\tilde{\mathcal{X}}$, IV)          ▷ features $\tilde{\mathcal{X}}$ sorted descending on IV
2: $\mathcal{X}^* \leftarrow \emptyset$                                ▷ selected features
3: for $\tilde{X}$ in $L_{\tilde{\mathcal{X}}}$ do                              ▷ feature Selection
4:     if IV($D_{disc}[\mathcal{X}^* \cup \{\tilde{X}\},, \mathcal{Y}]$) > IV($D_{disc}[\mathcal{X}^*, \mathcal{Y}]$) then
5:         $\mathcal{X}^* \leftarrow \mathcal{X}^* \cup \{\tilde{X}\}$
6: $\bar{\mathcal{X}}^* \leftarrow$ sortDesc($\mathcal{X}^*$, #B)          ▷ sort $\mathcal{X}^*$ in descending order on #B
7: return $\bar{\mathcal{X}}^*$
```

the model based on the training data. More features also result in a larger index size (number of attributes for the prediction table), which affects lookup performance and storage size. To address this problem, we select only the most significant features $\mathcal{X}^*$ for inclusion in the prediction table / index, optimizing for the predictive power of the selected set $\mathcal{X}^*$. We use information value (IV) as a metric for measuring the predictive power of a set of features. Once a set of features $\mathcal{X}^*$ has been selected, we determine the sort order for these features to be used in the index / prediction table. This step aims to reduce the index size as the sort order $\bar{\mathcal{X}}^*$ for the selected features $\mathcal{X}^*$ can affect the index size.

Feature Selection. The naive approach for selecting a set of features is to enumerate all $2^{|\tilde{\mathcal{X}}|}$ subsets of $\tilde{\mathcal{X}}$, compute the information value IV($D_{disc}[\tilde{\mathcal{X}}']$) of the projection of the discretized training data for each subset $\tilde{\mathcal{X}}'$, and then select a subset of features with the highest information value. While this guarantees that an optimal subset of features will be selected, the exponential number of queries (in $|\tilde{\mathcal{X}}|$) required is prohibitive. We opt for a linear greedy heuristic (Algorithm 1) that iterates over the features sorted decreasingly on their IV value. Starting from an empty set of features, for each feature $\tilde{X} \in \tilde{\mathcal{X}}$, the algorithm tests whether the information value for the set of features selected so far will be improved by including $\tilde{X}$ in the selection (Line 4). In this case, the algorithm adds $\tilde{X}$ to the set of selected features $\mathcal{X}^*$.

Preprint

Sorting Features. The quality of the approximation of the predictions of model f provided by our approach and the size of the prediction table only depends on the set of selected features. However, the size of an index built on top of the prediction table and the size of the index for the standalone version of our approach can be affected by the ordering of features in the index key.

Example 4 (Feature Order Affects Index Size). *Figure 5 shows an example of two trie indexes built over different sort orders for the same set of discretized features $\mathcal{X}^* = \{X_1, X_2\}$ (same IV score). Feature $\tilde{X_1}$ has five bins and $\tilde{X_2}$ has two bins. Assume for the sake of the example that all combinations of bins exist in D_{train}. If we put $\tilde{X}_1$ first in the ordering, we get a root node with five elements, each of which points to a node with two elements (for $\tilde{X}_2$). Building an index with this ordering requires six nodes and 15 values. If we first put $\tilde{X}_2$, we get three nodes and 12 values. This difference in size increases when there is a more significant disparity between the number of bins.*

In Algorithm 1 (Line 6), we use a greedy heuristic to determine the sort order $\bar{\mathcal{X}}^*$ to be used in the prediction table and index by sorting the features in $\mathcal{X}^*$ in decreasing order by their number of bins (recall that #B denotes the number of bins).

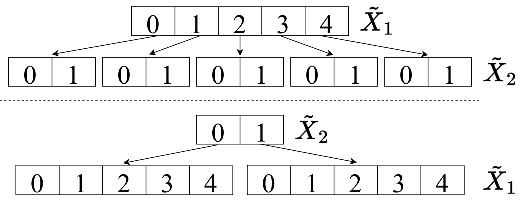


Figure 5: Two tries of different sizes over the same features.

4.3 Prediction Table and Index Population

Once we have determined the ordered list of features $\bar{\mathcal{X}}^*$ used as keys for prediction, we create the prediction table with schema $(\mathcal{X}^*, \mathcal{Y})$ and an index over this table using a standard index structure. We discuss our implementation of the standalone index version of our approach in Python in Section 4.4.

Key Creation. We map each data point $x \in D_{train}$ and its prediction $\hat{y}$ into the embedding space by determining to which bin each selected feature X^* in x belongs. Recall that we use δ^* to denote the mapping function. InferDB uses δ^* during inference, and we describe how to implement it in SQL in Section 5.1.

Aggregation. Next, we determine the prediction associated with each key x^* over $\mathcal{X}^*$ for which at least one training data point x exists such that $\delta^*(x) = x^*$. For that, we aggregate the predictions of all training data points D_{x^*} with the same key x^*

$$D_{x^*} = \{x \mid \delta^*(x) = x^*\}$$

using an aggregation function α suitable for the prediction task. We show definitions for three possible versions of α below. For regression tasks, α computes the mean prediction for all these training data points. For classification tasks, the predicted probabilities (picking the class with the highest sum of probabilities across these training data points) or the majority class among the predictions for the training data points will be used, as shown below. We use $p(x, y)$ for data point x and class y to denote the probability assigned by the model for x to belong to class y.

$$\alpha(D_{x^*}) = \frac{\sum_{x \in D_{x^*}} \hat{y}}{|D_{x^*}|} \qquad \text{(regression)}$$

$$\alpha(D_{x^*}) = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} |\{x \mid x \in D_{x^*} \land \hat{y} = y\}| \qquad \text{(majority vote)}$$

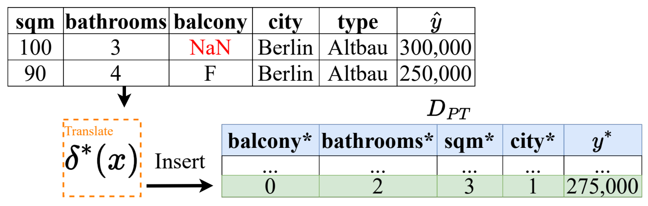
$$\alpha(D_{x^*}) = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \sum_{x \in D_{x^*}} p(x, y) \qquad \text{(maximum probability class)}$$

Insertion. We create the prediction table D_{PT} with schema $(\bar{\mathcal{X}}^*, \mathcal{Y})$:

$$D_{PT} = \{(x^*, \alpha(D_{x^*})) \mid \exists x \in D_{train} : \delta^*(x) = x^*\}$$

Example 5 (Prediction Table). *Figure 6 shows the process of creating a prediction table by first embedding each data point in D_{train} into $\mathcal{X}^*$ using δ^* and then inserting a new record with the aggregated prediction for D_{x^*} into the table.*

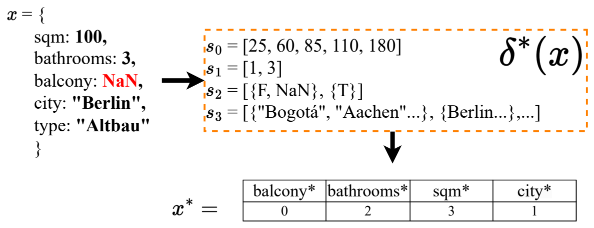
Runtime Complexity. Creating the prediction table is linear in the size of the training data. For each training data point x, we can compute $x^* = \delta^*(x)$ in linear time in the number of features in $\mathcal{X}^*$ and constant time in the maximum number of bins for the



sqm	bathrooms	balcony	city	type	$\hat{y}$
100	3	NaN	Berlin	Altbau	300,000
90	4	F	Berlin	Altbau	250,000

balcony*	bathrooms*	sqm*	city*	y^*
...	...	...	...	...
0	2	3	1	275,000

Figure 6: Building a prediction table for a regressor that predicts property price. Predictions for data points with the same discretization are aggregated and inserted into D_{PT}.



```
x = {
    sqm: 100,
    bathrooms: 3,
    balcony: NaN,
    city: "Berlin",
    type: "Altbau"
}
```

$\delta^*(x)$
s_0 = [25, 60, 85, 110, 180]
s_1 = [1, 3]
s_2 = [{F, NaN}, {T}]
s_3 = [{"Bogotá", "Aachen"...}, {Berlin...},...]
$x^* =$

balcony*	bathrooms*	sqm*	city*
0	2	3	1

Figure 7: Embedding a test data point using δ^* using the bins for each discretized feature in $\mathcal{X}^*$.

features in $\mathcal{X}^*$.[2] Furthermore, we must aggregate the predictions for each generated key x^*. The mean value for regression can be computed in linear time in $|D_{x^*}|$. The same applies to the majority vote and maximum total probability computation (by maintaining a hashmap from the values of the categorical feature to a count/sum). As each training data point exists in exactly one subset D_{x^*}, the total runtime of aggregating predictions is linear in $|D_{train}|$. As we assume the predictions of the model on the training data to be available as inputs, the overall runtime of creating a prediction table is in $O(k \cdot N)$ where $k = |\mathcal{X}^*|$ and $N = |D_{train}|$. Furthermore, note that k is typically a small constant. The runtime of building an index over the prediction table depends on the type of index structure we use, e.g., $O(k \cdot N)$ for a hash index.

4.4 Standalone Index in Python

While there are many benefits to storing our data structure in a database and using query processing for inference, our approach can also be implemented using a standalone index structure. As a proof of concept, we implement a trie data structure in Python to record the aggregated prediction for every discretized training data point. The nodes of the trie are implemented as dictionaries mapping discretized values for one or more features (tries use prefix compression) to a child node (intermediate nodes) or aggregated prediction (leaf nodes). Implementing our index structure in a more efficient language like C++ would be possible. Still, we choose Python here for a more fair comparison as Python frameworks are widely used for prediction and preprocessing [39, 43]. Our choice of a trie data structure is based on the fact that lookup in a trie is $O(k)$ (for $k = |\mathcal{X}^*|$) independent of the size of D_{train} and that common prefixes are compressed in a trie which is beneficial for sparse indexes (Section 5.3).

5 INDEX-BASED INFERENCE

This section presents how to use a prediction table for inference. In Section 5.1, we discuss transforming test data points into the embedding space (features $\mathcal{X}^*$) using SQL. Afterward, we demonstrate how to predict the transformed test data points by joining them with the prediction table in Section 5.2. Finally, we discuss how to deal with sparsity in the prediction table, i.e., how to use the prediction table for predictions for test data points x for which their key x^* in $\mathcal{X}^*$ after embedding does not exist in the prediction table.

[2]For categorical features, we use a hashmap to store the mapping between categorical values and bins.

Analog to the inference pipeline we are approximating with the prediction table, we assume as input a set of test data points D_{test}, which are stored in a database table. We are interested in optimizing the inference latency t_{index} using the prediction table, which is the sum of the runtime for computing the embedding for the test data points (t_{embed}) and the runtime of the join between the test data and the prediction table (t_{lookup}): $t_{index} = t_{embed} + t_{lookup}$.

5.1 Embedding Test Data Points

Consider the example from Figure 1 again. We want to predict the price of a single test data point x with features $\mathcal{X}$. To embed x, we must implement δ^* to bin the values for each feature $X \in \mathcal{X}$ for which the corresponding discretized feature X^* is in $\mathcal{X}^*$.

Example 6. *Figure 7 shows how data point x is mapped into the embedding space using δ^*. For each feature $X \in \mathcal{X}$ such that $X^* \in \mathcal{X}^*$, we use the bins for X to compute $x^*[X^*]$. For instance, $x[sqm] = 100$ is in the fourth bin for this feature ($[85, 110]$) and, thus, $x^*[sqm^*] = 3$.*

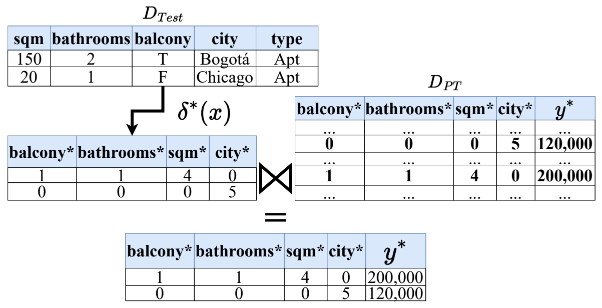
Numerical Features. For a numerical feature X, discretization partitions the domain of the feature $\mathcal{X}_i$ into $k = \#B_i$ bins with lower and upper bounds $\{b_{ij} = [l_{ij}, u_{ij}]\}$. To discretize a value v of feature X, we have to determine r such that $v \in b_{ir}$ (as the bins form a partitioning of the domain of X there exists exactly one r for any v). There are several options for how to implement this in SQL, e.g., storing the bin boundaries sorted on their bounds in an array and implementing a UDF that does a binary search over this sorted array, creating a mapping table $v \longrightarrow r$ (only feasible if the number of distinct values of X is low, e.g., exam grades), or using SQL's `CASE` statement to search through the bins linearly. As $\#B_i$ is typically small (often less than 10 bins are sufficient), we implemented the last option as it avoids the overhead of a UDF call and a join with a mapping table. As the bin boundaries are contiguous, it is sufficient to compare the value v against each bin's upper bounds (inclusive) when the bins are sorted in ascending order of their upper (or equivalently lower) bounds. The code generated for feature X_i is:

```
CASE WHEN $X_i$ <= $u_{i1}$ THEN 1 ... WHEN $X_i$ <= $u_{ik-1}$ THEN k-1
     ELSE k END AS $\tilde{X}_i$
```

Categorical Features. For categorical features X_i, each bin b_{ij} is a set of values. We store the bin membership in a mapping table $v \longrightarrow r$ for features with many distinct values. Otherwise, we use `CASE` statements using `IN` instead of inequalities:

```
CASE WHEN $X_i$ IN $b_{i1}$ THEN 1 ... WHEN $X_i$ IN $b_{ik-1}$ THEN k-1
     ELSE k END AS $\tilde{X}_i$
```

Preprint



sqm	bathrooms	balcony	city	type
150	2	T	Bogotá	Apt
20	1	F	Chicago	Apt

balcony*	bathrooms*	sqm*	city*
1	1	4	0
0	0	0	5

balcony*	bathrooms*	sqm*	city*	y^*
...	...	...	...	...
0	0	0	5	120,000
...	...	...	...	...
1	1	4	0	200,000
...	...	...	...	...

balcony*	bathrooms*	sqm*	city*	y^*
1	1	4	0	200,000
0	0	0	5	120,000

Figure 8: Predicting the price of a property using the discretized data point and a prediction table.

Example 7 (Implementing δ^* in SQL). *Consider $\mathcal{X}^* = \{\tilde{a}, \tilde{b}\}$ where feature a is numerical (domain $[1, 20]$) and and b is categorical (domain $\{red, green, blue\}$) and the bins for these features are:*

$$B_a = \{[1, 4], [5, 9], [10, 20]\} \quad B_b = \{\{red, green\}, \{brown\}\}$$

The SQL query for embedding the test data points is shown below.

```
SELECT CASE WHEN a < 5 THEN 0 WHEN a < 10 THEN 1 ELSE 2
       END AS a,
       CASE WHEN b IN ['red','green'] THEN 0 ELSE 1
       END AS b
FROM dbtest;
```

Runtime Complexity. Assuming that the maximum number of bins b for each feature in $\mathcal{X}^*$ and the number of selected features $k = |\mathcal{X}^*|$ is bounded by some constant, the cost of embedding is in $O(1)$ for each data point. Thus, the overall runtime of embedding a test dataset with $N = |D_{test}|$ is linear ($O(N)$). If we consider b and k as inputs, we get an overall runtime of $O(k \cdot b \cdot N)$.

5.2 Inference as Join

Given D_{test} stored as a table in the database, inference with the prediction table amounts to an equi-join on $\mathcal{X}^*$ between D_{test} after discretization and the prediction table.

Example 8 (Inference with Joins). *Figure 8 shows the inference process for two test data points using δ^* and the prediction table D_{PT} for the example introduced in Figure 1. First, the tuples are discretized using δ^*. Then, the discretized test table is joined with the prediction table D_{PT} on $\mathcal{X}^*$ to compute the predictions.*

Runtime Complexity. Let $M = |\{x^* \mid \exists x \in D_{train} : \delta^*(x) = x^*\}|$, i.e., M is the number of distinct points generated by embedding all training data points or equivalently the size of prediction table. As before, let $N = |D_{test}|$. Then, the inference runtime after embedding is the cost of an equi-join on $\mathcal{X}^*$ over two tables of size N and M where $\mathcal{X}^*$ is a key in prediction table. Thus, the size of the join result is bound by $max(N, M)$. If the smaller of the two tables fits into memory, then the DBMS will most likely use a hash join with runtime $O(N + M)$. That is, unless N is sufficiently small such that an index nested-loop join with runtime $O(N \cdot \log_B M)$ (assuming a B-tree with branching factor B) or $O(N)$ (assuming a hash index) is more efficient (the better asymptotic behavior outweighs the significantly higher constant factor of an index lookup). If both tables do not fit into memory, then we get another multiplicative logarithmic factor using a partition hash-join or merge join with multiple merge phases. So far we assume that every lookup in prediction table is successful. We defer the discussion on how to deal with failed lookups caused by sparsity to the next subsection.

5.3 Dealing With Sparsity

One potential issue with our approach is the sparsity of the prediction table, i.e., if there are many data points in the selected feature space $\mathcal{X}^*$ that are insufficiently covered by training data. For such a data point x^*, we have no (or very few) training data points x for which $\delta^*(x) = x^*$ and, thus, have insufficient information to make an informed prediction for x^*. However, the discretization technique we employ tries to ensure that each bin of a discretized feature covers at least a certain fraction of training data points. Additionally, we do not use all discretized features but select a subset $\mathcal{X}^*$ that is predictive. It is unlikely that features in $\mathcal{X}^*$ are highly correlated with each other as once our heuristic has chosen a feature $\tilde{X_i}$, we are unlikely to include a feature $\tilde{X_j}$ that is highly correlated with $\tilde{X_i}$ as this would not lead to a significant improvement in prediction quality (IV). Furthermore, sparsity is only a problem if the test data covers a lot of sparse regions. This is only possible if a significant difference exists between the training and test data distributions. However, most ML models perform poorly when there is a significant distribution shift, i.e., it is likely that the original model we are approximating would already perform poorly on this test data.

Dealing with sparsity through aggregation. Even though sparsity is unlikely to be a problem, we developed a technique for making predictions on sparse regions should the need arise. To compute a prediction for a data point x over $\mathcal{X}$ for which $x^* = \delta^*(x)$ does not exist in our data structure, we aggregate the predictions of data points x_k^* that agree with x^* on a prefix of the selected features. We use $\mathcal{X}^*[i]$ to denote the first i features in $\mathcal{X}^*$. Recall that $x[\mathcal{X}]$ denotes the projection of x on $\mathcal{X}$. We use $\mathcal{I}(x^*) = \perp$ to denote that x^* is not in the prediction table. If $\mathcal{I}(x^*) = \perp$, then we identify the largest i such that there exists at least one data point x_k^* with $\mathcal{I}(x_k^*) \neq \perp$ and $x_k^*[\mathcal{X}^*[i]] = x^*[\mathcal{X}^*[i]]$. As shown below, we then aggregate (using α as during prediction table construction) the predictions for all keys in the prediction table that match this prefix to compute the prediction y^* for x^*. That is, we further aggregate the approximated predictions over a larger region of the feature space $\mathcal{X}^*$ using the same rationale that underlies our approach for approximating models.

$$y^* = \alpha(\{x_k^* \mid x_k^*[\mathcal{X}^*[i]] = x^*[\mathcal{X}^*[i]]\})$$

$$\textbf{for} \quad i = \operatorname*{argmax}_j \exists x_k^* : x_k^*[\mathcal{X}^*[j]] = x^*[\mathcal{X}^*[j]] \wedge \mathcal{I}(x_k^*) \neq \perp$$

Alternatively, to create a prediction for x^*, we could generate synthetic training data points x such that $\delta^*(x) = x^*$ and create a new entry for x^* by aggregating the predictions of the model f for these data points on the fly. Note that this would require several calls to the model f, which only pays off if we reuse the prediction for x^* in the future.

Preprint

6 RELATED WORK

This section presents related work on in-DBMS machine learning (ML). Earlier studies coupled DBMSs with data-mining UDFs [1, 37, 46, 48]. Recent research improved UDF performance using column stores and vectorized operations [44]. Another line of work transpiles Python code to SQL to create a relational representation of ML pipelines [26, 27, 50]. In contrast, InferDB replaces complex inference pipelines with a lightweight embedding and index lookup, resulting in significant performance benefits and tighter DBMS integration.

Another important line of work extends the DBMS with native operators for training and inference [4, 11, 14, 53]. These approaches need more support for complex customized ML transformations as they require implementing new operators in the DBMS. Similar to these approaches, InferDB leverages native database operations. However, InferDB only uses technology that is readily available in all DBMS, i.e., it does not require new operators to be implemented inside the database, which is an essential advantage given the prevalence of one-off transformations in preprocessing [43] and the constant evolution of learning algorithms. Furthermore, in InferDB, users can write their processing, training, and inference pipelines using the tools they are already familiar with.

Approaches collocate DBMS engines and ML runtimes, e.g., Microsoft SQL Server/ONNX Runtime [33], Google BigQuery/TensorFlow [15], Amazon Redshift/SageMaker [2], and PostgresML Postgres/Scikit-learn [41]. This approach enables relational and ML operators to integrate into complex predictive analytics pipelines. Recent work explores cross-optimization opportunities in such pipelines [39]. Similarly, InferDB leverages optimization opportunities involving predictions in analytical pipelines such as predicate and aggregate push-downs and index/table scans.

Recently, there has been work on enhancing inference performance for XGBoost [6], LightGBM [25], and Scikit-learn [51] by rewriting pipeline code to use vector instructions. Recent research [21] in the brain-computer interface community has reported improvements of around 12x when using Intel's oneAPI data analytics library (oneDAL) [36] that uses AVX-512 instructions on Intel's hardware to optimize ML pipelines.

Our work resembles similarity (nearest-neighbor) search queries and k-nearest-neighbor (kNN) classification and regression [7, 16]. At inference time, kNN methods find the k nearest neighbors based on a similarity score. Then, a majority vote (classification) or other aggregate of the target feature (regression) determines the prediction. To accelerate neighbor search, kNN models build k-d trees [3] or ball trees [35] to organize the dimensions of the training instances. The number of training instances, N, dominates the complexity of querying a k-d tree $O(N\ log\ N)$ [7]. More efficient (approximate) methods are known, e.g., many approximate kNN approaches utilize locality-sensitive hashing [8, 18, 38, 40]. The right choice of k determines the generalization ability of kNN methods [52]. For small k, kNN methods tend to overfit the training data [52]. InferDB uses similarity-based predictions, but instead of storing training data and finding neighbors at inference time, it discretizes the feature space and creates an index with the $|\mathcal{X}^*|$ most relevant features, storing aggregate predictions for $|x^*|$ unique combinations of $\mathcal{X}^*$ selected features. Querying InferDB's index has a complexity of $O(|\mathcal{X}^*|)$ if using a trie, $O(log\ |\mathcal{X}^*|)$ if using a b-tree or $O(1)$ if using a hash index. Since, typically, $N >> |\mathcal{X}^*|$ and $N >> |x^*|$, having a structure independent of the number of training instances is very beneficial for performance. With this, InferDB avoids neighbor search at runtime and, even more importantly, leverages the generalization ability of more complex preprocessing pipelines to prevent overfitting to the training data. As we demonstrate in Section 7.6, this, in combination with the feature selection and discretization techniques employed by InferDB, leads to significant improvements in prediction accuracy compared to kNN.

7 EXPERIMENTAL EVALUATION

We now evaluate the performance of InferDB and compare it against baselines considering different setups, tasks, datasets, and models.

Setup & Workloads. We evaluate InferDB on a server with an AMD-EPYC-7742 2.25GHz CPU (10 cores used), 80 GiB RAM, and 8x3.2 TB SAS SSDs (RAID5). We use two datasets per task (regression, classification, multi-label classification), have designed pipelines, and have performed hyperparameter tuning for each. For all tasks, we evaluate common [43] types of models: linear regression models (LR), neural networks (NN), nearest neighbor (k-NN) models, and decision tree-based models, including gradient boosting frameworks XGboost [6] and LightGBM [25]). We measure the error using root mean squared log error (RMSLE) [23] for regression tasks and recall, precision, and F1 scores for classification tasks. **New York City Taxi Trip Duration (NYC-rides)** [22] contains 1.5 million taxi rides with 10 categorical and numerical features. Based on a Kaggle competition [22], we predict the trip duration (regression task). We design a pipeline based on the competition's top-performing solutions. **PM2.5 Concentrations (Pollution)** [12] is a time series dataset with 106 million records (daily pollution estimates) and 9 features. We use lagged and moving average features to predict the mean concentration (regression task). We use different train/test splits to evaluate the scalability of our solution (Section 7.6). **Credit Card Fraud (Fraud)** [42] contains $284,807$ credit card transactions. The task is to predict fraudulent transactions (binary classification). Only 0.172% of the instances are fraudulent, making the classification problem challenging. The dataset contains 29 numerical features. **Hit Songs (Hits)** [32] contains $142,963$ songs. The task is to predict if a song is a hit based on play count (classification task). The dataset contains 413 numerical and categorical features from acoustic properties and song metadata. **Rice Varieties (Rice)** [5] contains $75,000$ images of rice (106 numerical features). The task is to classify the rice type (multi-label classification, 5 classes). **MNIST (Digits)** [29] contains $70,000$ images of handwritten digits (multi-label classification, 10 classes). The images are 28x28 pixels (784 features). We use 70% of the data for training for all data sets unless stated otherwise.

7.1 Evaluated Systems

We compare the standalone version of InferDB with an inference pipeline using Scikit's implementations of preprocessing and learning algorithms. For the in-database version, we deploy InferDB in Postgres and compare it against a pure SQL implementation of the

preprint

Table 1: Average training and single instance inference latency with stddev. (5 runs) for the ML pipeline and InferDB (standalone).

Dataset	Model	ML Training Runtime [s]			ML Pipeline	InferDB Training Runtime [s]				InferDB
		Preprocessing	Learning	End-to-End	$t_{inference}$ [ms]	$t_{learn-embed}$	$t_{feature-sel}$	$t_{populate}$	$t_{build-index}$	t_{index} [ms]
NYC-rides	LR	998 ± 10	1.0k ±13	2.0k ±10	621 ±479	1.00 ± 0.2	2.0 ± 0.1	14.0 ± 2.0	17.0 ± 2.0	8.4 ± 0.04
	NN	996 ± 14	1.4k ±8	2.4k ±10	597 ±472	0.90 ± 0.2	2.8 ± 0.2	14.3 ± 0.8	18.0 ± 1.0	8.4 ± 0.04
	LGBM	998 ± 13	1.0k ±7	2.0k ±10	617 ±479	0.98 ± 0.2	4.6 ± 0.1	17.0 ± 0.5	22.5 ± 0.6	8.3 ± 0.00
Fraud	LR	0.4 ± 0.01	4 ± 0.1	4.4 ± 0.1	13 ± 1	0.6 ± 0.1	1.2 ± 0.2	13 ± 1	15 ± 1.3	0.02 ± 0
	NN	0.4 ± 0.00	9 ± 0.5	9.3 ± 0.4	20 ± 1	0.6 ± 0.2	3.0 ± 1.7	14 ± 2	18 ± 4.0	0.02 ± 0
	LGBM	0.3 ± 0.00	7 ± 5.0	7.0 ± 5.0	50 ± 1	0.6 ± 0.2	4.3 ± 0.3	17 ± 1	22 ± 1.2	0.02 ± 0
Hits	LR	78 ± 0.7	79 ± 0.7	156 ± 1.5	40 ± 2	7 ± 0	404 ± 3	524 ± 4.0	935 ± 6.0	0.27 ± 0
	NN	80 ± 4.0	101 ± 9.0	181 ± 11.0	42 ± 1	7 ± 0	403 ± 1	524 ± 3.0	934 ± 4.0	0.27 ± 0
	LGBM	6 ± 1.0	26 ± 3.0	32 ± 3.0	51 ± 7	7 ± 1	407 ± 2	527 ± 0.6	941 ± 1.7	0.28 ± 0
Digits	LR	2.4 ± 0.3	$1.8k$ ± 25	$1.8k$ ± 25	18 ± 0.1	1.9 ± 0.2	717 ± 4	795 ± 8	1.5k ±11	0.18 ± 0
	NN	2.3 ± 0.1	590 ± 155	592 ± 155	20 ± 2.0	2.0 ± 0.4	718 ± 14	799 ± 16	1.5k ±30	0.19 ± 0
	LGBM	2.1 ± 0.1	99 ± 8	101 ± 8	14 ± 3.0	2.0 ± 0.2	712 ± 5	790 ± 5	1.5k ±10	0.17 ± 0

inference pipeline, an implementation using PostgresML (PGML 2.0) [41] a machine learning extension for Postgres.

Prediction Table and Index. We store the prediction table in Postgres, and, unless stated otherwise, we use trie indexes. In Section 7.6 we compare several index structures.

SQL Model Following [26, 50], we generate SQL code for the preprocessing operators. We create a table storing the model's parameters. For LR, the parameter table has two columns: an id for the feature (primary key) and the feature's weight. For the NN, we use the approach of Schüle et al. [49] to transform the weight matrix of each layer into a relational representation. We create a table with four columns: a layer identifier l, an identifier i for a neuron (unique within the layer), an identifier j for a neuron in the next layer, and the weight w_{ij} of the connection between two neurons i (at layer l) and j (at layer $l + 1$). We use id $i = -1$ to store the biases. The primary key of this table is (l, i, j). To compute a prediction for LR, we join the preprocessed data with the parameter table to perform the dot product between the data point features and the weights. For NNs, we join the preprocessed data with the NN's parameter table and perform the subsequent activations.

PGML integrates Scikit's preprocessing operators and algorithms into Postgres. PGML reduces data transfer and duplication between the database and ML runtime operating on a shared memory space.

7.2 End-to-End Training Runtimes

Table 1 shows the breakdown of the training runtime for the ML pipelines and InferDB on four datasets. InferDB's time to create an index ($t_{build-index}$) is dominated by $t_{populate}$, which, in turn, is affected by three factors: the number of selected features, the number of unique keys after discretization, and the number of data points to aggregate. For the Digits dataset, the runtime for $t_{populate}$ and $t_{feature-sel}$ is almost the same. This is because the Digits dataset has many features (780), and InferDB selects a relatively large number of features (32), resulting in many unique keys. For Fraud, InferDB creates less than 8 bins for each of the selected features. Thus, the index population only requires aggregating around 140k predictions by the distinct observed combinations (~200) of the selected features $\mathcal{X}^*$ in D_{disc}. The classes in Fraud are highly imbalanced, resulting in few distinct combinations representing changes in the prediction function f.

Table 2: F1, Recall (R), Precision (P), RMSLE (Err) of InferDB and top-3 most effective ML models across different datasets

Dataset	Model	ML Pipeline				InferDB			
		F1	R	P	Err	F1	R	P	Err
NYC-rides	LGBM	-	-	-	0.37	-	-	-	0.46
	XGB	-	-	-	0.38	-	-	-	0.45
	NN	-	-	-	0.40	-	-	-	0.45
Pollution	XGB	-	-	-	0.27	-	-	-	0.27
	LGBM	-	-	-	0.27	-	-	-	0.28
	DT	-	-	-	0.42	-	-	-	0.58
Fraud	XGB	0.94	0.91	0.98	-	0.90	0.90	0.90	-
	LGBM	0.94	0.93	0.94	-	0.90	0.90	0.90	-
	NN	0.92	0.90	0.93	-	0.90	0.90	0.90	-
Hits	XGB	0.98	0.98	0.98	-	0.97	0.97	0.97	-
	LGBM	0.98	0.98	0.98	-	0.97	0.97	0.97	-
	DT	0.97	0.97	0.97	-	0.97	0.97	0.97	-
Digits	XGB	0.98	0.98	0.98	-	0.70	0.70	0.70	-
	LGBM	0.98	0.98	0.98	-	0.70	0.70	0.70	-
	LR	0.91	0.91	0.91	-	0.70	0.70	0.70	-
Rice	XGB	0.99	0.99	0.99	-	0.94	0.94	0.94	-
	LGBM	0.99	0.99	0.99	-	0.94	0.94	0.94	-
	LR	0.99	0.99	0.99	-	0.94	0.94	0.94	-

Preprint

7.3 Standalone Index Inference Runtime

Table 1 also shows the average inference latency for a single data point for the inference pipeline and InferDB. InferDB (t_{index}) outperforms the inference pipeline ($t_{inference}$) by ~ 2, ~ 3, and ~ 2 orders of magnitude for NYC-rides, Fraud, and Digits, respectively. Because of the complex preprocessing pipeline, The inference latency for the ML pipeline for the NYC-rides dataset is significantly higher than for Fraud and Digits.

7.4 Effectiveness

Table 2 shows the prediction accuracy achieved by the competitors. InferDB achieves nearly equivalent accuracy to the three most accurate ML pipelines for the Pollution, Hits, and Rice datasets. For Fraud and NYC-rides, InferDB's results are competitive with a slight loss in accuracy compared to the ML pipelines. However, InferDB is significantly less accurate for the Digits dataset. This is because InferDB selects 32 features to create the index, leading to a sparse embedding space. When no match for a datapoint is found,

Table 3: Average inference latency and size of InferDB and most effective ML model for different datasets in Postgres (5 runs)

Dataset	D_{test}	Model	ML Inference Latency Breakdown[ms]			ML	InferDB Inference Latency Breakdown[ms]			InferDB
			$t_{preprocess}$	$t_{predict}$	$t_{inference}$	Size[MB]	t_{embed}	t_{lookup}	t_{index}	Size[MB]
NYC-rides	438k	LGBM	11k ±119.00	26k ± 5k	38k ± 5k	13.00	450 ±28.0	2k ±43.00	2.5k ±33.0	13.00
Pollution	13M	XGB	12.6k ±146.00	473k ± 6k	486k ± 6k	1.08	4k ±87.0	4k ±53.00	8k ±75.0	0.07
Fraud	85k	XGB	70 ± 0.86	4k ± 5	4k ± 5	0.61	65 ±19.0	42 ±13.00	108 ±32.0	0.21
Hits	14k	XGB	183 ± 17.00	1.2k ±318	1.4k ±335	5.50	42 ± 0.5	7.8 ± 0.25	50 ± 0.7	0.04
Digits	7k	XGB	93 ± 21.00	2k ± 65	2.1k ± 84	7.60	25 ± 1.0	3.8k ± 8.00	3.8k ± 9.0	9.12
Rice	22k	XGB	124 ± 2.00	1.3k ± 4	1.4k ± 5	0.60	28 ± 0.4	82 ± 0.20	110 ± 0.6	0.40

predictions are aggregated on-demand as explained in Section 5.3. The sparse index also affects InferDB's performance, as will be discussed in Section 7.6. We elaborate on InferDB's current limitations to support sparse unstructured data tasks in Section 7.7.

7.5 Database Implementation

Storage Size. Table 3 reports the sizes of InferDB's prediction table (and its index) and the most effective ML pipelines. InferDB's prediction table consumes less space than the ML pipelines except for the XGBoost model for the Digits dataset and the LightGBM model for the NYC-rides, which have similar storage consumption to InferDB. This is because for the sparse input of Digits (28x28 pixels/784 features), InferDB selects 32 features to create the index. This results in over fifty million keys in InferDB's prediction table. On the other hand, for the Hits dataset, where InferDB selects only two features out of 413, the prediction table and its index consume only 0.05 MB. In comparison, the LightGBM ML pipeline uses ~ 140x more storage due to the large number of estimators (decision trees) required to achieve high effectiveness.

Batch Inference. Table 3 shows a breakdown of batch inference latency for InferDB and the most effective ML pipelines. InferDB consistently outperforms the competitors on all evaluated datasets except for Digits. This is because of the additional aggregation cost for on-demand predictions for data points not in prediction table. When InferDB chooses a small subset of features to create the index, the embedding dominates the end-to-end inference latency, as is the case for the Hits and Fraud datasets where InferDB selects only 2 out of 413 and 4 out of 30 features, respectively. For the ML pipelines, latency is dominated by the prediction step. This is because of the relatively large number of estimators for the XGBoost and LightGBM models. InferDB predicts 13M records in approximately 8 seconds while XGboost takes around 500 seconds. More importantly, as shown in Table 2, InferDB achieves almost the same accuracy as the best model for the Pollution dataset. In summary, InferDB consistently outperforms the competitors except when the prediction table is too sparse.

7.6 Micro-Benchmarks

Sparsity Analysis. We next analyze the impact of sparsity on the performance of InferDB. The volume of the embedding space grows exponentially in the number of features in $\mathcal{X}^*$. Thus, adding more features to the index introduces sparse regions, i.e., data points x^* where $\mathcal{I}(x^*) = \perp$. Sparsity can affect the performance of InferDB when the index misses the keys of test data points. Consider a test data point x with $x^* = \delta^*(x)$. As explained in Section 5.3, when $\mathcal{I}(x^*) = \perp$, we fall back to compute predictions *on demand*. For that, we perform a prefix search to find all keys that share the longest prefix with x^* that exists in the index and then compute the prediction for x^* on the fly using the aggregating function α.

Figure 9 shows the impact of adding more features to the index for the NYC-rides dataset. For this experiment, we added features based on IV. As more features are added to the index, the size of the embedded space grows exponentially (limited by the training data size). To measure the sparsity of the index and its impact on prediction over the test data, we define the *fill-factor* (out of all possible data points in the embedding space how many are covered by the index) and the *test-miss-rate* (the fraction of test data points x such that $\mathcal{I}(\delta^*(x)) = \perp$) as:

$$\text{fill-factor} = \frac{\# \, Keys \; in \; index}{\# \, Possible \; keys}$$

$$\text{test-miss-rate} = \frac{\sum_{x \in D_{test}} \mathbb{1}[\mathcal{I}(\delta^*(x)) = \perp]}{|D_{test}|}$$

Preprint

The *fill-factor* decreases as the embedded space size increases, and a smaller fraction of all possible data points in the embedding space are observed during training. Note the drop in *fill-factor* when adding a fourth feature to the index. This is because the features with higher predictive power (IV) also have more bins.

The *test-miss-rate* remains low until 7 features are added. Although the *fill-factor* is low with six features in the index, the sparse regions are rarely queried, confirming our suspicion that sparsity is less of a problem if training and test distributions are similar. In summary, InferDB's discretization and careful selection of features create a compact index where sparse regions are rarely queried. Nonetheless, even if sparsity becomes an issue, we can gracefully switch to prefix search to provide *on-the-fly* predictions for keys not in the index.

Index Type Selection. We measure inference latency for index types natively supported by Postgres: B-tree, hash index, and tries using SP-GiST [13]. We vary the size of the prediction table and the size of the test dataset D_{test}. The index types have no significant performance difference if the prediction table is not too sparse (Section 5.3). Figure 9.b shows the prediction latency (only the prediction step) for 1,000 test data points using different index types varying the size of the prediction table. The choice of index is only relevant if the *test-miss-rate* is high, as the index can perform the prefix search and return a prediction by aggregating the predictions of data points that share the same prefix. As shown in Figure 9.a, the trie index performs better than the B-tree and hash index for prefix search. The trie's page layout optimizes key collocation of keys with

Table 4: Avg. prediction latency ($\pm$ standard deviation) varying test/prediction table size and join algorithms (5 runs)

Features in index ($\lvert D_{PT} \rvert$)	$\lvert D_{test} \rvert$ (% of $\lvert D_{PT} \rvert$)	Hash join P.latency [ms]	Index-nested loop join P.latency [ms]
6 (13,853 paths)	1 (< 0.001%)	–	0.06 ± 0.004
	10 (< 0.01%)	–	0.41 ± 0.05
	100 (0.7%)	4.01 ± 1.21	3.37 ± 0.69
	1,000 (7%)	21.12 ± 6.1	19.08 ± 3.2
10 (217,929 paths)	1 (< 0.001%)	–	0.32 ± 0.04
	100 (0.04%)	–	3.2 ± 0.12
	10,000 (4.6%)	$2.8k \pm 152$	$2.7k \pm 18$
	100,000 (46%)	$27k \pm 340$	$26k \pm 180$

the same prefix, making it efficient to fetch all instances that share the same prefix. On the other hand, a prefix search with a B-tree can span several pages because its page layout is not optimized for prefix search. If the sparsity is high, i.e., low *fill-factor* and high *test-miss-rate*, a prefix search may span several pages, resulting in lower performance as shown in Figure 9.

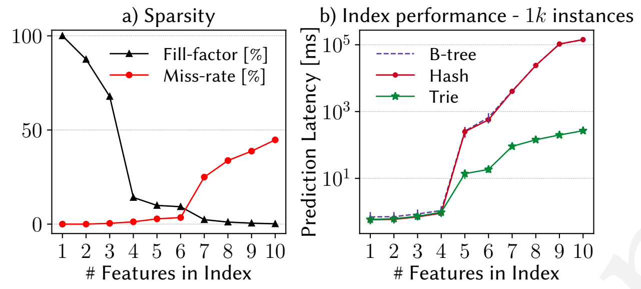


Figure 9: Fill-factor and test-miss-rate for an increasing number of features in the index and average prediction latency for 1,000 instances using different indexes.

Prediction Table and Test Table Sizes. We test the performance and usage of the index on prediction table compared to a full table scan on D_{PT} by varying the sizes of D_{test} and D_{PT} for the NYC-rides dataset. Table 4 shows the average inference latency under different query plans. Even if D_{PT} is small ($\sim$10k records), the index is only used when D_{test}'s size is less than 1% of D_{PT} because of the higher per tuple cost of index lookup compared to a full table scan. To compare the performance of a full table scan with an index scan when D_{test} is large and D_{PT} is small, we forced the use of the index (Postgres's *CPU index tuple cost* and *random page cost* parameters). As shown in Table 4, the performance ranges overlap, but the mean inference latency when scanning the index is slightly better than when scanning D_{PT}. If D_{PT} is large enough, an index scan is used when D_{test}'s size is 5% of D_{PT}'s size or less. As shown in Table 4, even when D_{test} is around 50% of D_{PT}, the performance is better when using the index. In summary, the index-nested loop join plan is faster even for small prediction and test tables.

Feature Selection. We compared our greedy search feature selection solution with a brute-force search on the Fraud dataset, which has 29 numerical features. The size Ω of the search space for possible candidates for $\mathcal{X}^*$ consists of all possible permutations of k features (index depth is k) for $k \in [1, |\mathcal{X}|]$. Thus, $\Omega = \sum_{k=1}^{29} \frac{29!}{(29-k)!} > 10^{30}$.

Table 5: Feature selection strategy performance

Search strategy	Selected features	IV	Size [B]	Runtime [s]
Greedy	$\{3, 9, 15, 16\}$	7.7	63,516	0.8
Brute-force	$\{3, 15, 16, 2\}$	7.9	50,868	13.4k

Table 6: Avg. inference latency, std. dev. and error for InferDB and LightGBM for different D_{test} in Postgres (5 runs)

D_{test}	Method	Inference Latency[s]	RMSLE
13M	PGML-LightGBM	484 ± 6	0.27
	InferDB	8.5 ± 0.04	0.28
26M	PGML-LightGBM	985 ± 16	0.26
	InferDB	20 ± 0.6	0.28

Therefore, we constrained the search space to the maximum index depth found by the greedy search algorithm, which resulted in a constrained search space of $\Omega_k = \sum_{k=1}^{4} \frac{29!}{(29-k)!} = 592,789$ permutations. The table shows the solutions' time (in seconds) and quality (IV score) for the two search strategies. Both select the third feature as the root for the index, but the greedy search finds a competitive solution at a significantly lower cost than the brute-force search. The solutions share 3 out of 4 selected features, explaining their similar IV values. The difference in storage consumption is due to the greedy solution including a feature with four bins (feature 9), while the brute force includes one with only two bins (feature 2). The greedy search shows a good tradeoff between predictive power and feature selection time.

Using Inference in Analytical Queries. We analyze InferDB's integration with inference-based queries using a query that filters prediction results for the NYC-rides dataset using a threshold on the trip duration (prediction result) on the NYC-rides dataset. This kind of query can be relevant when investigating the outcomes of a model for a specific subset of the data, e.g., *For which records in NYC-rides's D_{test} does the model predict a trip duration of less than x seconds.* Pushing down filters on database predicates can benefit query plans involving join-based inference in InferDB. This is impossible when using a model for predictions since we need to compute the predictions before filtering them. In Figure 10, the average inference latency for predicting 350k instances in the NYC-rides dataset is shown, with varying selectivity on D_{PT} by changing the threshold on the predicted trip duration. Competitors that compute predictions for all data points and filter afterward do not benefit from a more restrictive filter. Thanks to predicate push-down, when the selectivity of D_{PT} is around 1%, InferDB shows a 15% decrease in inference latency compared to a selectivity of 99%. In summary, InferDB enables analytical queries that involve prediction to benefit from standard query optimization techniques.

Scalability. In this experiment, we varied the size of D_{test}. We measured the inference latency and error for the Pollution dataset to evaluate InferDB's performance and effectiveness in handling large data volumes. We used one year's (2016) worth of data (around 26 million records) to train the model. Then, we measured inference latency and error on datasets containing six months' worth of data (13 million records) and one year's worth of data (26 million records).

Preprint

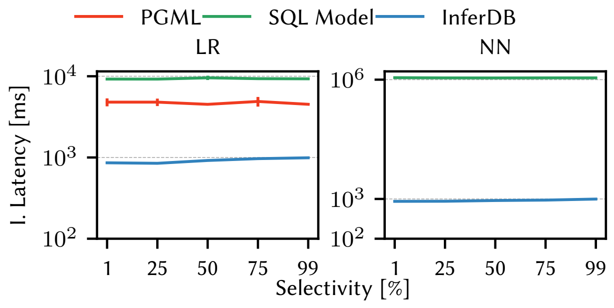


Figure 10: Average inference latency (± standard deviation) for predicting 350k instances of the NYC-rides dataset varying the query's selectivity on D_{PT} (3 runs)

Table 7: RMSLE of InferDB, kNN and LightGBM for the NYC-rides dataset

	LightGBM	InferDB	InferDB-training-labels	kNN	kNN-sel
RMSLE	0.38	0.46	0.6	0.69	0.57

The results shown in Table 6 demonstrate that InferDB achieves almost the same accuracy as a complex ML model (LightGBM) while delivering two orders of magnitude better performance.

Generalization. We compare InferDB against a version of our system that stores training labels instead of model predictions (InferDB-training-labels) and kNN using all features or the same features selected by our approach (kNN-sel). The idea is to evaluate the impact of storing a model's predictions in the prediction table rather than directly using the training data labels. We use the NYC-rides dataset, known for containing outliers in the trip distances and duration and missing values in the augmented traffic features. We report the RMSLE for a complex ML pipeline with the LightGBM regressor as a baseline. As shown in Table 7, InferDB outperforms kNN. This is due to InferDB's feature discretization and selection, which effectively groups similar data points for more accurate and stable prediction outcomes. InferDB is most effective when aggregating predictions made by a model on preprocessed data. This takes advantage of complex ML pipelines that generalize and avoid overfitting. Incorporating a preprocessing pipeline and a model reduces the impact of outliers and low-quality data on InferDB's predictions.

7.7 Discussion

InferDB consistently outperforms ML pipelines for regression, binary classification, and multi-label classification tasks. It achieves two orders of magnitude improvement in inference latency on average compared to state-of-the-art ML runtimes such as XGboost [6] and LightGBM [25] while preserving the ability to approximate the predictions of complex ML models with a low loss in accuracy. However, as demonstrated in Table 2 and Table 3, InferDB's performance and effectiveness are affected by sparsity. A larger embedding space can decrease performance and prediction accuracy by reducing the chances of finding an index entry for a test data point. This is especially problematic for computer vision tasks and high-dimensional text embeddings. In our tests, InferDB had the most significant accuracy drop in the digit recognition task. We also do not consider feature correlations, resulting in suboptimal selection when highly correlated features have high IV values.

In future work, we will rethink important aspects of InferDB to address these challenges. For instance, we may allow for a certain amount of feature engineering in the feature selection step, e.g., allow polynomial feature transforms. We could also generate synthetic training data points on the fly using the model's predictions as labels to create prediction table entries for missing keys when requested. This would allow our index to adapt to user requests over time.

InferDB's current design does not exploit that pipelines and prediction tasks may share common data or preprocessing steps. To exploit inter-index preprocessing, the discretization of features involved in different pipelines needs to consider the correlations of the involved features with the different targets. Finding an appropriate discretization that can be shared across multiple indexes can be difficult. To reduce storage consumption, InferDB's design should consider feature transformations, as mentioned above, that effectively preserve predictions in the discretized space.

We defer maintenance of our data structures under updates to future work. We envision three scenarios requiring different update strategies: (1) If the ranges of the bins remain the same, we can maintain aggregates incrementally and monitor distribution drift. (2) If the bin ranges change, we must invalidate affected entries and recompute the aggregates, but we can do so on a partial copy to reduce downtime. (3) If there's a significant data drift, we must retrain the model from scratch.

8 CONCLUSIONS

We present InferDB, a novel framework for approximating an end-to-end inference pipeline using a lightweight transformation of the data into an embedding space and indexing an approximation of predictions using standard index structures. This approach is easy to integrate into database systems such as Postgres, enabling the seamless combination of SQL queries with inference. Our approach significantly improves inference performance without significant accuracy loss. Future work includes maintaining structures under distribution shifts, adaptive index construction (the index is populated over time based on inference requests), and feature transformations in feature selection.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (ref. 414984028), the European Union's Horizon 2020 research and innovation programme (ref. 957407), and is in part supported by NSF awards IIS-1956123 and IIS-2107107.

preprint

REFERENCES

[1] Rakesh Agrawal and Kyuseok Shim. 1996. Developing Tightly-Coupled Data Mining Applications on a Relational Database System. In *SIGKDD*. 287–290.

[2] Amazon. 2020. Create, train, and deploy machine learning models in Amazon Redshift using SQL with Amazon Redshift ML. Retrieved April 11, 2024 from https://aws.amazon.com/de/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml/

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[4] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.

[5] İlkay Çınar and Murat Koklu. 2022. Identification of Rice Varieties Using Machine Learning Algorithms. *Journal of Agricultural Sciences* 28, 2 (2022), 307 – 325. https://doi.org/10.15832/ankutbd.862482

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *SIGKDD* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[7] T. Cover and P. Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13, 1 (1967), 21–27.

[8] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. 2011. Fast locality-sensitive hashing. In *SIGKDD*. 1073–1081.

[9] James Dougherty, Ron Kohavi, and Mehran Sahami. 1995. Supervised and Unsupervised Discretization of Continuous Features. In *ICML*. 194–202.

[10] Usama M. Fayyad and Keki B. Irani. 1993. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. In *IJCAI*. 1022–1029.

[11] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for In-RDBMS Analytics. In *SIGMOD*. 325–336.

[12] Centers for Disease Control and Prevention. 2023. Daily Census Tract-level PM2.5 concentrations, 2016-2020. Retrieved April 11, 2024 from https://healthdata.gov/dataset/Daily-Census-Tract-Level-PM2-5-Concentrations-2016/k9st-jhz8/data

[13] Walid G. and Ihab F. Ilyas. 2001. SP-GIST: An extensible database index for supporting Space Partitioning Trees. *Journal of Intelligent Information Systems*, 215–240.

[14] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesus Camacho-Rodriguez, and Matteo Interlandi. 2023. The Tensor Data Platform: Towards an AI-centric Database System. *CIDR* (2023).

[15] Google. 2023. Make predictions with imported TensorFlow models. Retrieved April 11, 2024 from https://cloud.google.com/bigquery/docs/making-predictions-with-imported-tensorflow-models?hl=de

[16] Gongde Guo, Hui Wang, David Bell, Yaxin Bi, and Kieran Greer. 2003. KNN model-based approach in classification. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. Springer Berlin Heidelberg, 986–996. https://link.springer.com/chapter/10.1007/978-3-540-39964-3_62#citeas

[17] Isabelle Guyon and André Elisseeff. 2003. An Introduction to Variable and Feature Selection. *J. Mach. Learn. Res.* 3, null (2003), 1157–1182.

[18] Omid Jafari and Parth Nagarkar. 2021. Experimental Analysis of Locality Sensitive Hashing Techniques for High-Dimensional Approximate Nearest Neighbor Searches. In *ADC*, Vol. 12610. 62–73.

[19] Andy Jassy. 2018. AWS re:Invent 2018 keynote. Video. Retrieved April 11, 2024 from https://www.youtube.com/watch?v=ZOIkOnW640A

[20] Harold Jeffreys. 1946. An invariant form for the prior probability in estimation problems. In *Proceedings of the Royal Society A*. Royal Society. https://doi.org/10.1098/rspa.1946.0056

[21] Kannadasan K, Haresh M V, Ambati Rami Reddy, and B. Shameedha Begum. 2023. BCIRecog: An Optimized BCI System for Imagined Speech Recognition. In *ICCCNT*. 1–7. https://doi.org/10.1109/ICCCNT56998.2023.10308091

[22] Kaggle. 2017. New York City taxi trip duration. Retrieved April 11, 2024 from https://www.kaggle.com/c/nyc-taxi-trip-duration

[23] Kaggle. 2017. New York City taxi trip duration evaluation. Retrieved April 11, 2024 from https://www.kaggle.com/competitions/nyc-taxi-trip-duration/overview/evaluation

[24] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. 2016. ExploreKit: Automatic Feature Generation and Selection. In *ICDM*. 979–984.

[25] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *NIPS* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 3149–3157.

[26] Steffen Kläbe and Stefan Hagedorn. 2021. Applying Machine Learning Models to Scalable DataFrames with Grizzly. BTW 2021. , 195–214 pages. https://doi.org/10.18420/btw2021-10

[27] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Exploration of Approaches for In-Database ML. In *EDBT*. OpenProceedings.org, 311–323. https://openproceedings.org/2023/conf/edbt/paper-7.pdf

[28] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567.

[29] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[30] Wei-Chao Lin and Chih-Fong Tsai. 2020. Missing value imputation: a review and analysis of the literature (2006–2017). *Artif. Intell. Rev.* 53, 2 (feb 2020), 1487–1509. https://doi.org/10.1007/s10462-019-09709-4

[31] Huan Liu, Farhad Hussain, Chew Lim Tan, and Manoranjan Dash. 2002. Discretization: An Enabling Technique. *Data Min. Knowl. Discov.* 6, 4 (oct 2002), 393–423. https://doi.org/10.1023/A:1016304305535

[32] Maximilian Mayerl, Michael Vötter, Günther Specht, and Eva Zangerle. 2023. Pairwise Learning to Rank for Hit Song Prediction. In *BTW 2023*. Gesellschaft für Informatik e.V., Bonn, 555–565. https://doi.org/10.18420/BTW2023-26

[33] Microsoft. 2023. Predict (transact-SQL) - SQL machine learning. Retrieved April 11, 2024 from https://learn.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-ver15

[34] Guillermo Navas-Palencia. 2020. Optimal binning: mathematical programming formulation. abs/2001.08025 (2020). arXiv:2001.08025 [cs.LG]

[35] Stephen M Omohundro. 1989. *Five balltree construction algorithms*. International Computer Science Institute Berkeley. https://omohundro.files.wordpress.com/2009/03/omohundro89_five_balltree_construction_algorithms.pdf

[36] Intel Oneapi-Src. 2020. Oneapi-src/onedal: Oneapi data analytics library (onedal). Retrieved April 11, 2024 from https://github.com/oneapi-src/oneDAL?tab=readme-ov-file

[37] C. Ordonez. 2006. Integrating K-means clustering with a relational DBMS using SQL. *TKDE* 18, 2 (2006), 188–201.

[38] Jia Pan and Dinesh Manocha. 2011. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *SIGSPATIAL*. 211–220.

[39] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-End Optimization of Machine Learning Prediction Queries. In *SIGMOD*. 587–601.

[40] Yongjoo Park, Michael J. Cafarella, and Barzan Mozafari. 2015. Neighbor-Sensitive Hashing. *PVLDB* 9, 3 (2015), 144–155.

[41] Postgresml. 2022. Postgresml/postgresml: PostgresML is an AI application database. Download open source models from Huggingface, or train your own, to create and index LLM embeddings, generate text, or make online predictions using only SQL. Retrieved April 11, 2024 from https://github.com/postgresml/postgresml

[42] Andrea Dal Pozzolo, Olivier Caelen, Reid A. Johnson, and Gianluca Bontempi. 2015. Calibrating Probability with Undersampling for Unbalanced Classification. In *SSCI*. 159–166.

[43] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* 51, 2 (2022), 30–37.

[44] Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, and Stefan Manegold. 2018. Deep Integration of Machine Learning Into Column Stores. In *EDBT*. OpenProceedings.org, 473–476. https://doi.org/10.5441/002/EDBT.2018.50

[45] Maximilian Rieger, Moritz Sichert, and Thomas Neumann. 2022. Integrating deep learning frameworks into main-memory databases. In *Proceedings of the VLDB 2022 Applied AI for Database Systems and Applications Workshop co-located with (VLDB 2022) (AIDB Workshop Proceedings)*. https://drive.google.com/file/d/1GfZH3Y1sQKgplnnpTEM_E4skWdhmyrfe/edit

[46] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. 1998. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. In *SIGMOD*. 343–354.

[47] Iqbal H. Sarker. 2021. Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN Comput. Sci.* 2, 3 (mar 2021), 21. https://doi.org/10.1007/s42979-021-00592-x

[48] Kai-Uwe Sattler and Oliver Dunemann. 2001. SQL Database Primitives for Decision Tree Classifiers. In *CIKM*. 379–386.

[49] Maximilian Emanuel Schüle, Alfons Kemper, and Thomas Neumann. 2023. NN2SQL: Let SQL Think for Neural Networks. In *BTW*, Vol. P-331. 183–194.

[50] Maximilian E. Schüle, Luca Scalerandi, Alfons Kemper, and Thomas Neumann. 2023. Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL. In *EDBT*. 40–52.

[51] Scikit-learn. [n.d.]. 8.2. computational performance. Retrieved April 11, 2024 from https://scikit-learn.org/stable/computing/computational_performance.html

[52] Shichao Zhang, Xuelong Li, Ming Zong, Xiaofeng Zhu, and Debo Cheng. 2017. Learning k for KNN Classification. *ACM Trans. Intell. Syst. Technol.* 8, 3, Article 43 (jan 2017), 19 pages. https://doi.org/10.1145/2990508

[53] Yuhao Zhang, Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *PVLDB* 14, 10 (2021), 1769–1782.