Matryoshka: Uncovering Relevant Features in Data Lakes to Enhance Machine Learning Applications

Fedor Turchenko
BIFOLD & TU Berlin
Berlin, Germany
fedor.turchenko@tu-berlin.de

Runjie Zhang
UC San Diego
San Diego, USA
ruz039@ucsd.edu

Binger Chen
Matthias Boehm
BIFOLD & TU Berlin
Berlin, Germany
chen@tu-berlin.de
matthias.boehm@tu-berlin.de

Babak Salimi
UC San Diego
San Diego, USA
bsalimi@ucsd.edu

Amir Shaikhha
University of Edinburgh
Edinburgh, United Kingdom
amir.shaikhha@ed.ac.uk

Ziawasch Abedjan
BIFOLD & TU Berlin
Berlin, Germany
abedjan@tu-berlin.de

ABSTRACT

Data coverage and completeness are crucial prerequisites for machine learning (ML) applications. Data discovery allows retrieving relevant features from unstructured data lakes to augment the training data. However, existing methods either target only individual ML tasks, decouple table retrieval from feature selection, or neglect inter-feature relationships, resulting in redundant, high-dimensional data. In this paper, we introduce Matryoshka, a holistic system that discovers and selects relevant features from data lakes for downstream ML applications. Matryoshka replaces costly join materialization with compact Gram matrix sketches and performs feature selection via incrementally fitted linear proxy models with correlation-based pruning, yielding redundancy-free feature sets for regression and classification. Matryoshka improves downstream prediction quality by 19.3% on average while achieving the lowest geometric mean runtime and up to 120x faster execution on join-intensive workloads.

PVLDB Reference Format:
Fedor Turchenko, Runjie Zhang,
Binger Chen, Matthias Boehm, Babak Salimi, Amir Shaikhha, and Ziawasch Abedjan. Matryoshka: Uncovering Relevant Features in Data Lakes to Enhance Machine Learning Applications. PVLDB, 19(11): 3704 - 3717, 2026.
doi:10.14778/3836663.3836719

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/D2IP-TUB/Matryoshka.

1 INTRODUCTION

Modern machine learning (ML) is largely data-centric, where quality often hinges on high-quality and good-coverage features rather than specific models [42, 43]. Data lakes offer a large source for augmenting datasets with external features, but discovering them is costly. A straightforward approach would materialize candidate joins, retrain the target ML model, and analyze feature importance to eliminate redundancy. At data-lake scale, however, this process quickly becomes prohibitive. For instance, a single 1.3 M-row query table may need to be joined with 20 tables producing roughly 1 000 candidate features.

Limitations of Existing Work: General-purpose dataset search engines retrieve potentially relevant tables from data lakes [2, 4, 28], but do not address downstream feature engineering and selection, leaving substantial manual effort to the user. Prior ML-driven discovery systems attempt to close this gap and follow one of two categories shown in Example 1. First, there are *custom indexes* that score each candidate feature independently, for instance, via pairwise correlation or mutual information [18, 20, 34, 35]. These methods are fast but ignore inter-feature dependencies due to their univariate nature and thus may return redundant, mutually correlated features. Second, *composed systems* rely on general-purpose retrieval engines [5, 6, 46, 47] for enumerating joinable tables and join paths, and then apply multivariate feature selection over the materialized results [7, 22, 25, 27]. Although this approach provides more flexibility, it incurs substantial costs of reading, pre-processing, and joining candidate tables, often having to resort to sampling, which loses information and may incorrectly rank candidates. When multiple rows join to a single base entity, both system categories typically fix the aggregation to `MEAN`, ignoring that the optimal choice is task- and feature-dependent.

Example 1 (Real-Estate Price Prediction). *Consider the Real-Estate price-prediction table from our benchmark (Section 7): It contains property listings per street address, to be augmented from the Canada/US/UK lake with 70.000 tables [14]. Figure 1 shows the query and three joinable lake tables, each containing useful real-estate signals, such as building permit information, the zoning and floor information, as well as fire safety sheets. Information, such as the building's floor area is, reported redundantly by both the permits and zoning tables. Category (1) scores each candidate column against the price independently, so it returns the duplicated floor area from both tables, yielding redundant and misleading features. Category (2) instead trains the model on a materialized join over 537 retrieved candidates, but joining every candidate at lake scale is infeasible in main memory, so it samples the query and fails to find useful features.*

Matryoshka Design: To overcome these limitations, we introduce Matryoshka, a dataset discovery system for ML applications

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836719

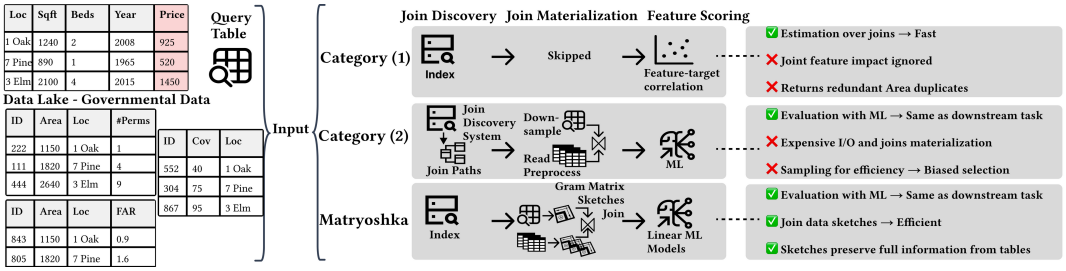


Figure 1: Approaches to ML-centric Dataset Discovery (Example 1).

based on the following design decisions: (1) pre-computing Gram matrix sketches per join key at indexing time for evaluating candidate features via ML proxy models without scanning raw lake tables or materializing joins, (2) filtering feature sets through multicollinearity-aware selection, (3) supporting regression and classification with a single index, and (4) fast greedy selection through incremental computation. To this end, we make the following **contributions**:

- *Feature-Selection-Aware Index* (Section 4): A dedicated inverted index for joinable dataset search that stores Gram matrix sketches per join key, enabling feature evaluation through ML proxy models without accessing raw tables.
- *ML-centric Discovery Operators* (Section 5): Operators for ranking retrieved features, selecting optimal aggregation functions for N-to-N joins, and performing multicollinearity-aware selection with respect to the downstream ML task.
- *Feature Selection Optimizations* (Section 6): Techniques for incremental learning and computation reuse that accelerate greedy feature selection over large candidate pools.
- *Experiments* (Section 7): We provide a comprehensive evaluation across several benchmarks, diverse proxy models, and a broad set of baselines, validating that Matryoshka remains effective and efficient under realistic data lake conditions.

2 PROBLEM FORMULATION

Let the query dataset be $Q=\{(\kappa_i,x_i,y_i)\}_{i=1}^{n}$, where $\kappa_i \in \mathcal{K}$ is an entity key, $x_i \in \mathbb{R}^d$ are base features, and $y_i \in \mathcal{Y}$ is the prediction target. Furthermore, a data lake is a collection of tables $\mathcal{D}=\{T_1,\ldots,T_{|\mathcal{D}|}\}$, and M denotes an ML pipeline, including pre-processing and training procedure. Here, $\mathcal{L}(M\,|\,S)$ is the estimated generalization loss of M on a dataset S, obtained by training M on training folds of S and evaluating on hold-out folds, e.g., predefined train/validation splits or K-fold cross-validation. Before defining the problem, we outline the table augmentation framework.

Cardinality-preserving Augmentation: Joining raw lake tables to Q can duplicate query table rows, e.g., N-to-N joins, altering the learning problem. We therefore restrict ourselves to *cardinality-preserving* augmentation: lake tables contribute only per-key features that are left-joined to Q. For each $T \in \mathcal{D}$, $\mathsf{Keys}(T)$ are the candidate key columns joinable to the query key domain $\mathcal{K}$, and

$$\mathcal{J}(Q,\mathcal{D}) \subseteq \{(T,k_T) : T \in \mathcal{D}, k_T \in \mathsf{Keys}(T)\}$$

is the set of joinable table-key pairs. Furthermore, $\mathcal{A}$ is a set of aggregation/encoding operators. For any $(T,k_T) \in \mathcal{J}(Q,\mathcal{D})$, column c of T, and $A \in \mathcal{A}$, there is a candidate feature function

$$f_{T,k_T,c,A}(u) = A(\{t[c] : t \in T, t[k_T] = u\}), \qquad u \in \mathcal{K}, \tag{1}$$

with handling of empty groups (missing value). If A is vector-valued, we treat each coordinate as a separate (scalar) candidate feature. Let $\mathcal{F}=\mathcal{F}(Q,\mathcal{D},\mathcal{A})$ be the set of all such candidate features induced by $\mathcal{J}(Q,\mathcal{D})$ and $\mathcal{A}$. For any subset $F \subseteq \mathcal{F}$, we define the augmentation of row i as $\phi_F(\kappa_i,x_i) = [x_i; z_F(\kappa_i)]$, with $z_F(\kappa_i) = [f(\kappa_i)]_{f\in F}$, and then write $Q \oplus F$ for the dataset obtained by replacing each row (κ_i,x_i,y_i) with $(\kappa_i,\phi_F(\kappa_i,x_i),y_i)$.

Definition 1 (Data Lake Feature Selection Problem). *Given $(Q,\mathcal{D},\mathcal{A})$, an ML pipeline M, a loss estimator $\mathcal{L}$, and a feature budget $s \in \mathbb{N}$, the* data lake feature selection *problem is*

$$F^* = \arg\min_{F\subseteq\mathcal{F}:|F|\le s} \mathcal{L}\big(M\,|\,Q\oplus F\big). \tag{2}$$

Finding F^* is computationally hard at data-lake scale. The candidate space $\mathcal{F}$ is large (tables × keys × columns × operators), and the feature utility is multivariate, depending on feature interactions. Concretely, for a lake of $|\mathcal{D}|$ tables each with at most K candidate join keys, C feature columns, and $|\mathcal{A}|$ aggregation operators, the candidate pool is of size $|\mathcal{F}| = O(|\mathcal{D}|\cdot K\cdot C\cdot|\mathcal{A}|)$. Enumerating all feature subsets of size at most s yields up to $\sum_{i=0}^{s}\binom{|\mathcal{F}|}{i}$ combinations, which is exponential in $|\mathcal{F}|$ for any fixed $s \ge 1$. Evaluating each candidate subset requires materializing the joins and retraining M. Both steps are infeasible at data-lake scale.

3 SOLUTION OVERVIEW

In this section, we first introduce the conceptual design of our feature selection approach, which is based on linear proxy models with incremental computation of model parameters. Subsequently, we discuss its integration in the system architecture of Matryoshka.

3.1 Linear Proxies for Feature Selection

To find the best feature set, the naïve approach is to materialize the join between the query table and all joinable lake tables and apply standard (greedy) feature selection algorithms, which trains and evaluates models on subsets of features. To reduce the computation costs, we resort to using lightweight proxy models that approximate the behavior of the final model to guide feature selection.

Linear Models and Covariance Joins: Good candidates for proxies are linear ML models, such as linear regression for regression tasks and linear discriminant analysis (LDA) for classification tasks, because they have closed-form solutions that can be efficiently computed for each subset. Additionally, we have to overcome the

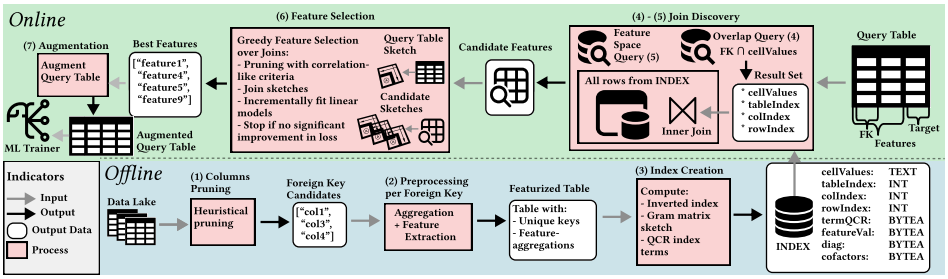


Figure 2: Matryoshka System Architecture Overview.

problem of materializing the join with possibly relevant data lake tables. Both linear regression and LDA rely on the covariance matrix to estimate the model parameters. In case of dataset discovery for ML, it is thus sufficient to materialize the join of the covariance matrix obtained from the query table and candidates from the data lake. Instead of joining two relations with n rows and m columns, we materialize the much smaller joins of $m \times m$ matrices. In Section 4.2, we discuss the Gram Matrix sketches, a dedicated data structure for computing covariance matrices over joins.

Application Scope: The linear proxies only rank candidate features rather than predict the target, a far weaker requirement than fitting it accurately, with linear regression for regression and LDA for classification tasks. As high-bias, low-variance estimators, they yield relevance rankings that are stable across data samples, so the selected features remain informative for flexible low-bias, high-variance downstream models, as we show experimentally in Section 7.2. Purely non-linear signals cannot be captured. Low-order polynomial sketches (Section 5.2) capture the univariate case, while richer non-linear selection does not reduce to our second-order statistics.

3.2 Matryoshka System Architecture

To augment a query table Q, Matryoshka discovers and selects features from a data lake $\mathcal{D}$ that minimize the validation loss $\mathcal{L}(M \mid Q \oplus F)$ of a downstream ML pipeline M. We leverage a pre-computed index containing Gram matrix sketches and an inverted index for joinability search, which facilitates training linear proxy models. The system comprises offline and online phases as shown in Figure 2.

Offline Phase: During the offline phase, we index the data lake by processing each table $T \in \mathcal{D}$. First, we prune columns unlikely to serve as join keys or useful features, such as those with many missing values, near-constant values, or text (step 1 in Figure 2). For each table, we identify potential foreign keys $\text{Keys}(T)$, while numeric and low-cardinality categorical columns are retained as candidate features. Second, for each candidate key $k_T \in \text{Keys}(T)$, we produce per-entity features via `GROUP BY` aggregation using max/min/mean/-median for numeric and counts for categorical features (step 2). This pre-aggregation ensures cardinality-preserving augmentation without row duplication in the query table. Third, we compute Gram matrix sketches for each aggregated feature table, consisting of distinct feature values, covariance terms, and a QCR [34] index term for correlation-based pruning (step 3). These sketches are stored together with an inverted index mapping cell values to table-column-row locations in a relational database for efficient online retrieval.

Online Phase: For a query table Q with join column k and prediction target y, we discover and select the feature subset $F^* \subseteq \mathcal{F}$ that minimizes the proxy model loss. The join column k is part of the query specification, and Matryoshka finds the best feature set with respect to this key. Using the inverted index, we look up each value from k via a two-step query, and select tables with sufficient overlap, ranked by join key coverage (step 4 in Figure 2). For the top-k joinable tables, we retrieve their Gram matrix sketches and assemble the candidate feature pool $\mathcal{F}$ with all necessary statistics for proxy model evaluation (step 5). Feature selection then proceeds in two phases (step 6): first, correlation-based pruning removes features with low predictive signal; then, greedy forward selection iteratively builds F^* by joining sketches and incrementally fitting linear proxy models, terminating when no candidate yields further improvements. The selected features F^* are used to augment the query table (step 7), producing $Q \oplus F^*$ for downstream ML model training.

4 INDEX

To enable feature selection over joinable tables retrieved from a data lake, the index has to support the following requirements.

R1 – Fast Joinable Tables Discovery: The index must support fast discovery of tables that join with the user's query table.

R2 – Effective Feature Extraction: The index must provide features in numeric format so they can be passed to feature selection as well as support aggregation to avoid N-to-M joins.

R3 – Efficient Feature Selection Support: The index must allow for fast retrieval of Gram Matrix sketches, and efficient feature selection without join materialization.

R4 – Maintainability: Finally, the index must allow updates.

4.1 Index Layout

We extend known inverted index structures [1, 20] (Section 4.1), by smoothly integrating additional signals, such as Gram matrix sketches [36] (Section 4.2), correlation sketches [34] into a unified index structure. We store the inverted index locations and the Gram matrix sketches together in one relational table, where each row, keyed by a table, column, and row identifier, carries both the cell value used to discover joinable tables and the per-key sketch used to select features over the join. A single structure thus answers both index queries through native relational operations, a grouped aggregation for joinable table discovery and an indexed lookup for sketch retrieval, serving features for both regression and classification.

Inverted Index Relational Model: We choose an RDBMS as the backbone of our index because it consists of cell value locations and Gram matrix sketches, which is a combination of relational and embedding data. We prefer RDBMSs over vector databases, which are not designed for querying extensive metadata. In our setting, the inverted index serves also positional metadata features embedded in the sketch. The index for an entire data lake can be large and RDBMSs provide the necessary scalability. The online protocol (Section 4.3) issues two relational queries against this table: a grouped aggregation for joinable table discovery and a composite-key inner join for sketch retrieval. A key-value store cannot execute grouped aggregations and relational joins without pulling all candidate rows into memory first. Lucene-style indices cannot efficiently retrieve multi-attribute numerical payloads by a composite document key. Transactional row-level updates during index maintenance further satisfy requirement R4.

Inverted Index Attributes: The location of a cell value is defined with table, column, and row identifiers as shown in Figure 2 (bottom right). Since we pre-aggregate raw features by performing `GROUP BY` aggregation over a foreign key candidate column, cell values stored in the index are unique within the scope of the column. This layout enables fast estimation of token overlap with a user query column to find joinable tables. We benefit from in-database indexes and query optimizations to facilitate fast joinable tables search. We create two B-tree indexes, one on the CellValues column for computing token overlap and one on a combination of TableIndex, ColIndex, and RowIndex for retrieving the Gram matrix sketches per row of joinable tables. Through effective pruning of the foreign key candidates and the systematic enumeration of aggregations, we apriori materialize the Gram matrix sketches for fast feature selection at runtime.

Sketch Attributes: To enable an efficient joint Gram matrix computation [36], we pre-compute the following aggregates:

- *Constant aggregate (count):* The count of tuples per join key, which is always one in case of pre-aggregation.
- *Linear aggregate (featureVal):* The values obtained after basic aggregation (min, max, mean, median).
- *Quadratic aggregate:* The outer product of featureVal. This matrix is decomposed into diagonal and off-diagonal elements, stored in the index diag and cofactors fields.

The sketches are stored compactly as PostgreSQL `BYTEA` objects. We achieve fast loading of these arrays in Polars dataframe using the Arrow Database Connectivity (ADBC) framework, which directly maps PostgreSQL `BYTEA` to Arrow column buffers.

4.2 Offline Pre-computation

Algorithm 1 details the entire offline phase of Matryoshka. The algorithm consists of three main steps. For each table in the data lake, (1) we prune unlikely foreign key columns and form the foreign key candidate column set. Subsequently, (2) we group by the candidate columns and compute per-key aggregated features. Finally, (3) we derive the Gram matrix sketch components for the resulting features and insert the table-column-row locations along with the sketch values into the index. In the following, we describe these steps in detail.

Step 1 – Columns Pruning: As the first step of Algorithm 1, we employ several simple heuristics to discard unlikely foreign keys. Columns with a cardinality lower than 1%, more than 80% missing values, i.e., `NULL`, or text, i.e., with an average length of more than 100 characters, are neither used in grouping nor as a part of Gram matrix sketches. We estimate the heuristics on up to 10,000 samples per column (Algorithm 1: Line 7).

Algorithm 1: Matryoshka Index ($\mathcal{D}$)

Input: Data lake $\mathcal{D}$, DB connection string db_conn.
Output: Relation "InvertedCofactorIndex" R.

```
1  R ← create_table(db_conn)
2  tableIndex ← 0 ;                                  // Inverted index attribute
3  for each table t ∈ D do
4      df ← read_table(t) ; tableIndex ← tableIndex+1
       // Step 1: Columns Pruning
5      FKC ← ∅ ;                                     // Set of foreign key candidates
6      for each column c ∈ df do
7          s ← random_sample(c, n=10_000) ;          // Sample 10,000 rows
8          stats ← compute_stats(s)
9          flag ← eval_heuristics(stats)
10         if flag is True then
11             FKC ← FKC∪c
       // Step 2: Aggregation and Feature Extraction
12     num_cols,cat_cols ← split_by_data_type(FKC);
         // Sets of column names
13     colIndex ← 0 ;                                // Inverted index attribute
14     for each column c ∈ FKC do
15         colIndex ← colIndex+1
16         num_table ← group_by_agg_num(t, c, num_cols\{c})
17         cat_table ← group_by_agg_cat(t, c, cat_cols\{c})
18         feature_table ← num_table ⋈_c cat_table
           // Step 3: Gram Matrix Sketch computation
19         rowIndex ← 0 ;                            // Inverted index attribute
20         for each row r ∈ feature_table do
21             rowIndex ← rowIndex+1
22             count,featureVal,diag,cofactors ← compute_sketch(r)
23             R ← insert_into_table(R,db_conn,tableIndex,
24                 colIndex,rowIndex,count,featureVal,diag,cofactors)
```

Step 2 – Aggregation and Feature Extraction: We meet the index requirement of *N*-to-*1* joins by performing a `GROUP BY` operation over each foreign key candidate column and aggregating numeric or categorical features with multiple aggregation functions (Algorithm 1: Step 2). This approach ensures flexibility during feature selection as one may not have prior knowledge of optimal aggregations of certain features for a given downstream ML task. Aggregating numeric columns is straightforward and we use functions such as max, min, mean, and median. This aggregation produces the first intermediate table *num_table* (Algorithm 1: Line 16). In contrast, computing various aggregations of categorical features is challenging. Additionally, we need to select the encoding method to convert data into numeric format. Since we aim at providing multiple variants of aggregated features for the downstream ML tasks, we propose a frequency encoding scheme, which we combine with mean, max, and non-zero count aggregation functions. This aggregation results in the second intermediate table *cat_table* (Algorithm 1: Line 17). Figure 3 shows an example of our frequency encoding. Suppose the "Status" column only has two unique values shown in the table. Then, after performing `GROUP BY` over the "ZIP" column, we compute frequencies of categories per-key, adding zero entries for missing categories. Once frequency vectors are obtained, we compute the aggregates over these.

Step 3 – Inverted Index and Gram Matrix Sketch: Having joined the intermediate tables with numeric and categorical features

ZIP	Status
111	New
333	New
444	Old

→

ZIP	Status
111	New
333	New
444	Old

Figure 3: Frequency Encoding Example.

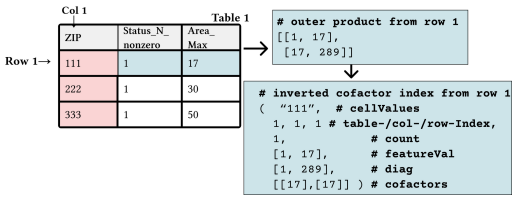


Figure 4: Gram Matrix Sketch and Inverted Index.

(Algorithm 1: Line 18), we need to define the inverted index attributes and compute the Gram matrix sketch for each of the rows (Algorithm 1: Step 3). "Table 1" in Figure 4 shows the example output of the join between the two intermediate tables (Algorithm 1: Line 18). For the sake of presentation, we show only a single aggregate for the numeric feature "Area" and categorical feature "Status". The column "ZIP" is a grouping column, i.e., a foreign key candidate. To compute the Gram matrix sketch, we calculate the outer product for each row. We then split the diagonal and off-diagonal elements of this matrix into two separate sketch attributes diag and cofactors, while raw values are written to the featureVal attribute. The table, column, and row indices are mapped to the location of a cell from the grouping column "ZIP" and are written to the dedicated inverted index attributes along with respective cell values.

4.3 Online Querying the Index

In the online phase, there are two main queries on the index.

Overlap Query: First, we find top-k joinable tables by running the overlap query OQ (Listing 1), which computes the intersection size of the query column with the cellValues attribute. A `GROUP BY` over the tableIndex and colIndex inverted index attributes aggregates cell values of potential foreign keys, which are then checked against the set of query column Q values.

Feature Space Query: Second, we retrieve the sketches of top-k joinable tables to perform the feature selection by running the feature space query FQ (Listing 2). The query retrieves all attributes from the index, but only for the value-column-row combinations of joinable tables. This process requires creating a temporary table from the OQ result set and then using an inner join to filter the index table. The OQ result set is unnested with respect to the cellValues and "rowIndices" attributes to generate combinations of inverted index terms for filtering as these columns were aggregated (lines 4 and 6 in Listing 1). The result set of this query is then used by feature selection.

4.4 Index Maintenance

Updating the index of Matryoshka involves identifying whether the table already exists in the index and calibrating the Gram matrix sketch. For numeric features, `MAX`, `MIN`, and `MEAN` aggregates are incrementally maintainable, while for `MEDIAN` aggregates, we can maintain a sketch like T-digest [16], which estimates quantiles

```
1  SELECT oq.cellVals, oq.tableIx, oq.colIx, oq.rowIndices, oq.nCells
2  FROM (
3    SELECT tableIx,colIx,COUNT(cellVals) AS nCells,
4      STRING_AGG(cellVals, '|') AS cellVals,
5      STRING_AGG(rowIx::text, '|') AS rowIndices
6    FROM InvertedCofactorIndex
7    WHERE cellVal IN (Q)
8    GROUP BY tableIx, colIx
9  ) AS oq
10 ORDER BY numberOfCells DESC
11 LIMIT K;
```

Listing 1: Overlap Query

```
1  DROP TABLE IF EXISTS overlapQueryResults;
2  CREATE TEMPORARY TABLE overlapQueryResults (
3    cellValues TEXT, tableIndex INT,
4    colIndex INT, rowIndex INT, numberOfCells INT);
5  COPY overlapQueryResults
         (cellValues, tableIndex, colIndex, rowIndex, numberOfCells)
6    FROM STDIN (FORMAT BINARY);
7  SELECT * FROM InvertedCofactorIndex
8    INNER JOIN overlapQueryResults
9    USING (tableIndex, colIndex, rowIndex)
10   ORDER BY (tableIndex, colIndex, rowIndex, cellValues);
```

Listing 2: Feature Space Query

from a summary of data. For categorical features, one can maintain a fixed-dimensional vector of per-category relative frequencies over the dictionary of categorical values, together with the total group count. Incoming records can be incorporated exactly by rescaling existing frequencies and updating a single category entry, without access to the raw data if categories already exist in the dictionary. Unseen categories necessitate schema expansion or other strategies. Aggregate descriptors derived from this frequency vector, such as `MAX` and `N_NONZERO`, are incrementally maintainable, while the `MEAN` is invariant due to normalization.

5 FEATURE SELECTION

We now discuss how Matryoshka navigates the candidate feature space $\mathcal{F}$. We bridge statistical robustness, computational efficiency, and scalability for ML-oriented data discovery by leveraging precomputed Gram matrix sketches. This way, we standardize features, apply ridge regularization, and rank candidates with robust model selection criteria, all computed over joins from a single index and accelerated by computation reuse and parallel evaluation.

5.1 Computing Gram Matrix Over Joins

During feature selection, we need to compute the Gram matrix of the joined query table and augmentation candidate without materializing the join. We build on factorized learning over joins [36]. The original approach computes the Gram matrix without materializing the join, but only for a single join that is known in advance and recomputed for every query. In a data lake, the joinable tables are not known upfront, and the same table participates in many queries. Matryoshka therefore moves this computation offline and precomputes each lake table's per-key sketch once, storing it in the index so the joint Gram matrix of a query table and any candidate can be assembled at runtime by a join on the key. The Gram matrix sketch join operator matches the query and candidate Gram matrix sketches on the join key and aggregates them into the joint Gram matrix $\mathbf{X}^\top\mathbf{X}$.

Example 2 (Gram Matrix Sketches Join). *Figure 5 shows an example of the Gram matrix join on sketches. The augmentation candidate*

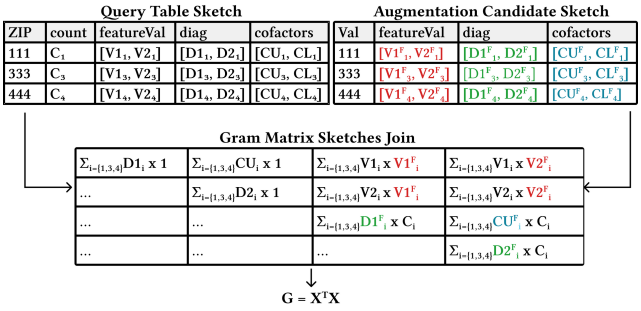


Figure 5: Gram Matrix Sketches Join.

sketch is the result set of the feature space query $FQ_{\Theta}(OQ(Q;K=1))$ *from Listing 2. The rows between the two sketch tables joined over columns ZIP and Val. Both the query table and the augmentation candidate contain two features in the sketch attributes.* C_i,V_{ij},D_{ij} *represent the instances of* COUNT, FEATUREVAL, *and* DIAG *attributes of the sketch where* j *is the feature identifier and* i *is the first character of the join column. The* COFACTORS CU_i *and* CL_i *denote the upper and lower off-diagonal elements of the outer product.*

Gram Matrix Construction: Computing the Gram matrix boils down to per-key summations and multiplications over the sketch:

- The diagonal entries are obtained by multiplying D_{ij} of one input with the corresponding C_i from the other input, and summing over matched instances. Counts for augmentation candidates are one as they are pre-aggregated per join key. The off-diagonal entries follow the same pattern.
- Within-table covariances, i.e., between query table features, are computed using the COFACTORS CU_i and scaled by the matching count from the other side.
- Cross-table covariances, between query and augmentation features, are obtained by multiplying their corresponding FEATUREVAL V_{ij} and V_i^F j and summing over all join keys.

Joining the two sketches yields $G = X^\top X$ in blocks indexed by table pairs: within-table blocks (query × query, cand × cand) on the diagonal, cross-table blocks (query×cand) off it. By symmetry, each diagonal block needs only one triangle; each cross block is computed in full. For Linear Regression, feature-target covariance $\mathbf{X}^\top \mathbf{y}$ is computed in the same fashion.

5.2 Feature Selection Algorithms

Using our Gram matrix join operator, we support the well-established greedy strategies of forward selection and backward elimination. Both evaluate candidate features through our proxy models.

Proxy Models and Scoring: Rather than retraining the downstream model at each iteration, we employ linear proxies. We use linear regression for regression and Linear Discriminant Analysis for classification, and evaluate them in closed form from the second moment sketches. We score each candidate feature by its incremental contribution beyond the already selected features. For regression, a ridge-regularized Schur complement gives the residual sum of squares that the candidate removes conditioned on the current set. We rank by the resulting generalized cross-validation score [12], so its complexity penalty curbs over-selection. For classification,

Algorithm 2: Forward Feature Selection in MATRYOSHKA

Input: Query table sketch, candidate sketches, threshold ϵ, task type.
Output: Selected feature set F_s.
// Initialize from query table sketch
1 $cov \leftarrow$ build_covariance(query_sketch); // per-class for LDA
2 $factorization \leftarrow$ decompose(cov); // ridge, for incremental reuse
3 $F_s \leftarrow \emptyset$
4 **while** $|candidates| > 0$ **do**
5 $best\_score \leftarrow -\infty$; $best\_f \leftarrow$ None
// Score candidate contributions via sketch join
6 **for** *each feature* f *in candidates* **do**
7 $joint\_cov \leftarrow$ join_sketches(query_sketch, sketch[f])
8 **if** *task is regression* **then**
9 $score \leftarrow -$conditional_gcv($factorization$, $joint\_cov$)
10 **else**
11 $score \leftarrow$ conditional_fisher($factorization$, $joint\_cov$)
12 **if** $score > best\_score$ **then**
13 $best\_score \leftarrow score$; $best\_f \leftarrow f$
14 $best\_cov \leftarrow joint\_cov$
15 **if** $best\_score - current\_score < \epsilon$ **then**
16 **break**; // No significant improvement
17 $F_s \leftarrow F_s \cup \{best\_f\}$; candidates $\leftarrow$ candidates $\setminus \{best\_f\}$
18 $current\_score \leftarrow best\_score$
19 $factorization \leftarrow$ update_factorization($factorization$, $best\_cov$)
20 **return** F_s

the analogous conditional Fisher ratio measures class separation using the Mahalanobis distance [29] and projects out the separation already captured by the selected features. The ridge terms keep both scores well defined when the joint Gram matrix is ill-conditioned, which is common on lakes with many near-collinear candidates.

Forward Selection: Algorithm 2 iteratively builds the feature set by adding one feature at a time. At each iteration, we evaluate all remaining candidates and select the one providing the largest utility improvement. In our data lake setting, however, each candidate must be joined with the query table, and lakes often exhibit many candidates, necessitating algebraic and computational optimizations. We first build the covariance matrix from the query table's Gram matrix sketch and compute its factorization for efficient incremental updates (lines 1–3). For regression, we compute a single covariance matrix, whereas for LDA, we maintain per-class covariance matrices. At each iteration, we join every candidate feature sketch with the query table sketch to obtain the joint covariance and score the candidate from it while reusing the committed factorization, so no candidate requires a separate fit (lines 6–14). The feature yielding the best improvement is selected, and the factorization is updated incrementally for the next iteration (lines 17–19). The algorithm terminates when no candidate provides further improvement above threshold ϵ.

Backward Elimination: Unlike forward selection, backward elimination in Algorithm 3 starts with all candidate features and iteratively removes the least useful ones. In an initialization phase, we assemble the joint covariance matrix similar to forward selection (lines 1–5). The factorization over the joint covariance matrix provides a convenient mechanism for eliminating collinear features before selection begins (lines 6–11). The diagonal entries $\mathbf{C}_{f,f}$ of the coefficient matrix from F-GS decomposition (Section 6) represent the residual variance of each feature after projecting out preceding features. Features with diagonal values below threshold τ are nearly

Algorithm 3: Backward Elimination in Matryoshka

Input: Query & candidate sketches, thresholds ϵ, τ, task type.
Output: Selected feature set F_s.

```
// Phase 1: Join all sketches and build full covariance
1  cov ← build_covariance(query_sketch)
2  for each candidate f do
3      cov ← extend_covariance(cov, join_sketches(query_sketch, sketch[f]))
4  factorization ← decompose(cov) ;            // ridge F-GS decomposition
5  F_s ← all candidates
   // Phase 2: Eliminate collinear features via diagonal analysis
6  C ← get_coefficient_matrix(factorization)
7  for each feature f in F_s do
8      if |C_{f,f}| < τ then
9          F_s ← F_s \ {f} ;                   // Remove collinear feature
10         cov ← remove_feature(cov, f)
11         factorization ← decompose(cov) ;                // Recompute
   // Phase 3: Greedy backward elimination
12 while |F_s| > 0 do
13     worst_score ← ∞;   worst_f ← None
14     for each feature f in F_s do
15         cov' ← remove_feature(cov, f)
16         factorization' ← decompose(cov')
17         if task is regression then
18             score ← gcv(factorization')
19         else
20             score ← −fisher_ratio(factorization')
21         if score < worst_score then
22             worst_score ← score;   worst_f ← f;   best_cov ← cov'
23     if worst_score − current_score > ϵ then
24         break ;                  // Removing any feature hurts too much
25     F_s ← F_s \ {worst_f}
26     cov ← best_cov ;                                  // Commit removal
27 return F_s
```

linear combinations of others and are removed, reducing the candidate pool and making the covariance matrix less singular. The greedy elimination phase (lines 12–26) then iteratively removes the feature with smallest performance degradation. Unlike forward selection, backward elimination does not benefit from incremental factorization updates. Each candidate evaluation requires removing the feature from the covariance matrix and recomputing the factorization. However, the candidate features pool for backward elimination is smaller than in forward selection due to collinearity elimination.

Parallel Evaluation: Each iteration of these algorithms allows two levels of parallelism. First, the candidate evaluations in the inner loop (Algorithm 2, Lines 6-14 and Algorithm 3, Lines 14-22) are independent: every candidate's sketch join, factorization update, and score computation read the committed state and produces a score. We partition the candidate pool into batches and dispatch them as parallel tasks. Each task takes the current Gram matrix and a batch of candidate features, evaluates these candidates, and returns the per-candidate scores. Second, the inter-iteration Gram matrix cache update is also parallelizable. After committing the best feature, we update remaining candidates' Gram matrix to incorporate cross-covariances with the newly selected feature. Each candidate's update reads the committed state and writes an independent cache entry.

Non-Linear Sketch Extension: The linear proxy captures pairwise covariances between candidate features and the query target but cannot model non-linear relationships. A natural mitigation is to integrate precomputed polynomial terms $x^2, x^3, \ldots, x^d$ into the sketch of the candidate column x. During forward selection, these terms are evaluated jointly with the base feature: each per-round fit incorporates the additional columns, enabling the proxy to detect multiplicative dependencies while preserving the algebraic structure of the incremental Gram matrix updates. The degree d is a user parameter that affects the per-round cost. For the LDA classification proxy, monotone power transforms leave the discriminant directions invariant up to rescaling and thus cannot alter the decision boundary.

5.3 Feature Space Pruning

Data lakes often have many candidate features with little predictive signal. Evaluating all candidates can be expensive. To address this issue, Matryoshka applies lightweight correlation-based pruning to filter out irrelevant features before selection.

Pruning for Regression: For regression tasks, we use the Quadrant Count Ratio (QCR) [34] index, which approximates Pearson correlation between each candidate feature and the target over joins. Our index supports dedicated hash-based attributes for QCR as well as Gram matrix sketches from which the covariance matrix and thus, the correlation ratio can be derived. The hashed terms required for QCR are precomputed during offline indexing, enabling efficient correlation estimation at query time. Candidates with QCR scores below a user-specified threshold are discarded before feature selection.

Pruning for Classification: For classification tasks, we use the correlation ratio η^2, which measures the proportion of variance in a feature explained by class membership as the ratio of between-class to total variance. Both quantities can be computed from the joint covariance matrix over joins. Features with correlation ratio below the threshold indicate weak class separability and are pruned.

6 OPTIMIZING GREEDY FEATURE SELECTION

On top of the parallel evaluation of candidates and pruning, we leverage incremental computation and reuse for runtime efficiency.

6.1 Incremental Sketch-based Selection

Evaluating candidate features naïvely requires materializing the join. To address this bottleneck, we leverage *incremental linear algebra approaches* that operate directly on Gram matrix sketches, including incremental QR decomposition, LU factorization, Cholesky decomposition, and conjugate gradient descent (CGD).

Factorized Gram-Schmidt (F-GS): Among these alternatives, we found that the Factorized Gram-Schmidt (F-GS) decomposition [39] provides the best fit for our framework due to its ability to: (1) operate purely on second-order statistics ($\mathbf{G} = \mathbf{X}^\top \mathbf{X}$) without requiring access to the full data matrix $\mathbf{X}$, (2) maintain numerical stability through carefully ordered orthogonalization steps, and (3) support efficient $O(k^2)$ per-candidate updates compared to $O(mk^2)$ for standard QR methods, where m is the number of rows and k is the number of features. The F-GS approach decomposes the Gram matrix $\mathbf{G}_k = \mathbf{X}_k^\top \mathbf{X}_k \in \mathbb{R}^{k \times k}$ into $\mathbf{G}_k = \mathbf{R}_k^\top \mathbf{R}_k$, where $\mathbf{R}_k$ is upper triangular and represents the Cholesky-like factor of the Gram matrix. Additionally, we maintain the coefficient matrix $\mathbf{C}_k = \mathbf{R}_k^{-1}$, which maps Gram space coordinates to orthonormal basis coefficients. This factorization enables us to efficiently compute projections and orthogonalizations needed for evaluating candidate features without ever forming the orthonormal matrix $\mathbf{Q}$ explicitly.

Incremental Gram Matrix Updates via F-GS: For a new candidate feature f_{k+1} with vector $\mathbf{x}_{k+1}$, we need to evaluate its impact on the model. From our index, we retrieve the cross-covariance sketch $\mathbf{g}_k = \mathbf{X}_k^\top \mathbf{x}_{k+1} \in \mathbb{R}^k$ (relationship between existing features and candidate) and the self-covariance $\gamma = \mathbf{x}_{k+1}^\top \mathbf{x}_{k+1} \in \mathbb{R}$ (candidate's variance). These sketches form the augmented Gram block:

$$\mathbf{G}_{k+1} = \begin{bmatrix} \mathbf{G}_k & \mathbf{g}_k \\ \mathbf{g}_k^\top & \gamma \end{bmatrix} \tag{3}$$

The incremental F-GS update proceeds by computing projection coefficients $\boldsymbol{\alpha}_k = \mathbf{C}_k^\top \mathbf{g}_k \in \mathbb{R}^k$, which represent how much the new candidate projects onto each existing orthonormal direction. We then construct the orthogonalized direction:

$$\tilde{\mathbf{c}}_{k+1} = \begin{bmatrix} -\mathbf{C}_k \boldsymbol{\alpha}_k \\ 1 \end{bmatrix} \in \mathbb{R}^{k+1} \tag{4}$$

and compute its norm in the Gram inner product space with $\rho = \sqrt{\tilde{\mathbf{c}}_{k+1}^\top \mathbf{G}_{k+1} \tilde{\mathbf{c}}_{k+1}}$. This norm represents the residual variance in the candidate feature after removing all components already explained by the current feature set. The updated factorization is then:

$$\mathbf{R}_{k+1} = \begin{bmatrix} \mathbf{R}_k & \boldsymbol{\alpha}_k \\ \mathbf{0}^\top & \rho \end{bmatrix}, \qquad \mathbf{C}_{k+1} = \begin{bmatrix} \mathbf{C}_k & -\mathbf{C}_k \boldsymbol{\alpha}_k / \rho \\ \mathbf{0}^\top & 1/\rho \end{bmatrix} \tag{5}$$

This update requires only $O(k^2)$ operations per candidate evaluation, compared to $O(k^3)$ for computing a fresh Cholesky decomposition, which would need the complete Gram matrix or CGD.

Incremental Least Squares Solution: For regression tasks, we similarly update the least-squares solution incrementally. We maintain the right-hand-side sketch $\mathbf{s}_k = \mathbf{X}_k^\top \mathbf{y}$ (the covariance between features and target) and retrieve $s_{k+1} = \mathbf{x}_{k+1}^\top \mathbf{y}$ for each candidate from the index. The transformed right-hand side is computed as:

$$\mathbf{d}_{k+1} = \mathbf{C}_{k+1}^\top \begin{bmatrix} \mathbf{s}_k \\ s_{k+1} \end{bmatrix} \tag{6}$$

The coefficients $\boldsymbol{\beta}_{k+1}$ of the augmented model are then obtained by solving the triangular system $\mathbf{R}_{k+1} \boldsymbol{\beta}_{k+1} = \mathbf{d}_{k+1}$ via backward substitution in $O(k^2)$ time. The last entry of $\mathbf{d}_{k+1}$ directly provides the projection of the target onto the new orthogonal direction, allowing to compute the exact RSS reduction: $\Delta\text{RSS} = \mathbf{d}_{k+1}^2$. This quantity serves as a reliable proxy for the candidate's utility without retraining.

6.2 Reuse During Feature Selection

We exploit computation reuse at two levels: *query table reuse*, where the query table's Gram matrix is computed once before selection, and *inter-iteration reuse*, where cross-covariances between selected features are preserved across iterations.

Query Table Reuse: As shown at the top of Figure 6, the query table's Gram matrix block remains constant throughout feature selection. Naïvely recomputing the joint Gram matrix for each candidate would redundantly aggregate the query table's diagonal entries ($D1_i$, $D2_i$), cofactors (CU_i), and within-table covariances. Instead, we compute the query table's self-covariance block once. For each candidate evaluation, we then compute only the cross-covariance terms between the query features and the candidate feature (e.g., $\sum_i V1_i V1_i^F$ or $\sum_i V1_i V2_i^F$) and the candidate's self-covariance.

Inter-Iteration Reuse: Figure 6 (bottom) also shows how computations are reused across iterations. After Feature 1 is selected in iteration 1, its cross-covariances with the query table features become part of the committed Gram matrix. When evaluating Feature 2 in the subsequent iteration, we do not recompute the entire joint Gram matrix from scratch. We reuse the already-computed 3×3 block (in gray). The new terms required are (1) Feature 2's covariances with the original query features ($\sum_i V1_i \times V2_i^F$ and $\sum_i V2_i \times V2_i^F$) and (2) the cross-covariance between Features 1 and 2. The cross-covariance of the two selected features is retrieved from the CU^F COFACTOR attribute of the sketch when both features originate from the same augmentation candidate table. This term was precomputed and indexed offline. Otherwise, we compute the cross-covariance of selected features as $\sum_i V1_i^F V2_i^F$. The F-GS factorization update then operates on this incrementally extended Gram matrix.

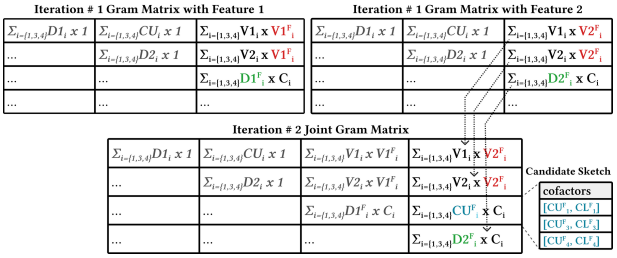


Figure 6: Computation Reuse during Forward Selection.

Complexity Reduction: This two-level reuse strategy significantly reduces the total sketch join operations. For a selection process that evaluates n candidates over K iterations, the naïve approach requires $O(n \cdot K)$ full Gram matrix computations. With reuse, the query table's self-covariance is computed only once, and cross-covariances between previously selected features are never recomputed.

7 EXPERIMENTS

We study MATRYOSHKA on various ML tasks, data lakes, and datasets against state-of-the-art baselines, complementing end-to-end experiments with micro-benchmarks.

7.1 Experimental Setup

Datasets and Evaluation Setups: We evaluate MATRYOSHKA for several public and real-world query datasets summarized in Table 1, covering regression and classification. As pre-processing, we drop rows with missing query keys or targets, impute remaining missing values (mode for categorical, iterative Bayesian Ridge for numerical), and apply AutoGluon's [17] feature generator to normalize types. For augmentation, we use the data lakes in Table 2. We assess MATRYOSHKA under three setups, each pairing different query tables and lakes. *(1) In-the-wild (ITW)* (Section 7.2) is the most realistic in two variants. For *external*, the query comes from outside the lake and is paired with a geographically (NYC, CUK) or topically relevant (Gittables) lake. For *in-lake*, the query is from the lake itself, as marked by *From Lake* in Table 1. *(2) AutoFeat* (Section 7.2) adopts the eight binary classification lakes of AutoFeat [27], where task-relevant features are put into auxiliary tables with hidden foreign key relationships, so a method must recover connections and discriminate useful features. *(3) Kitana* (Section 7.3) follows the leave-one-out

Table 1: Query Datasets (*From Lake* indicates if from same lake, top partition: regression, bottom partition: classification).

Dataset	Lake	Query col.	Target	Rows	Features	From Lake
Jobs	CUK	Region	Payscale	5,850	4	✓
IMDB [31]	Git	Title	Rating	5,000	4	×
Page Views [45]	Git	Name	Total Views	11,341	8	×
VG Sales [37]	Git	Publisher	Global Sales	16,598	5	×
Energy [9]	NYC	Block	GHG	3,374	3	×
Fire Incidents	NYC	ZIP Code	Duration	1,330,883	5	✓
Real Estate [33]	NYC	City	Price	35,768	7	×
Arrest Records	CUK	ZIP Code	Arrest Type	18,026	5	✓
Hospital	CUK	County	Substance	99,366	5	✓
Food Facts	Git	Group	Eco-Score	569,289	11	×
Street Trees	NYC	Street	Condition	484,607	7	✓

Table 2: Data Lakes (*%N/S* is ratio of numeric/string columns).

Lake	# Tables	# Rows		# Cols		%N/S	Raw
		median	*p90*	*median*	*p90*		
NYC [20]	1,070	170	53,959	11	35	46/54	52 GiB
CUK [14]	11,805	20,296	157,821	13	30	49/51	111 GiB
Gittables [26]	975,684	67	272	11	24	54/46	16 GiB
Kitana [25]	258	1,743	55,088	16	45	23/77	554 MiB

strategy of Kitana [25], where each annotated table in turn serves as the query and remaining tables form the augmentation pool.

Baselines: We compare Matryoshka with forward feature selection against five state-of-the-art baselines. *ARDA* [7] is an end-to-end augmentation system with coreset-based sampling, candidate evaluation against random noise, and exact and fuzzy joins via aggregation strategies (AutoFeat implementation [13]). *AutoFeat* [27] discovers transitive features over multi-hop join paths and ranks them by information-theoretic relevance and redundancy. *Kitana* [25] is a data-centric AutoML system with factorized learning, linear-proxy candidate evaluation, and greedy feature selection (authors' implementation [11]). *QCR* [34] finds joinable columns correlated with the target via a quadrant-based sketch index (Blend [20] implementation). *CAAFE* [24] iteratively prompts an LLM to generate feature engineering code and keeps features that improve a base classifier (authors' implementation [23] with Gemini 2.5 Flash Lite). CAAFE, ARDA, AutoFeat, and Kitana require join paths as input, for which we use LSH Ensemble [47] (LakeBench [14] implementation) with multi-threaded join graph construction on the large GitTables [26] lake. All baselines use default hyper-parameters.

Hardware and Software: We use a server with an AMD EPYC 7702P 64-core processor (128 logical threads) and 512 GB of DDR4 RAM, running Ubuntu Linux. Matryoshka is implemented in Python 3.12 and uses PostgreSQL as its backend, accessed via ConnectorX [44] for high-throughput reads. Numerical routines rely on NumPy [38], SciPy [41], and scikit-learn [32]. Polars [40] serves as our Dataframe engine, and candidate sketch construction and per-candidate proxy evaluation are parallelized with Ray [30].

7.2 Matryoshka Against Baselines

Setup: Because public query datasets are well-curated and leave little room for augmentation, the ITW setup simulates more realistic query tables. We define three simulation regimes of increasing information availability. *Join Key Only* retains only the join column, the most data-scarce setting. *Categorical Features* adds all original categorical columns. *Binned Features* retains all columns, with numeric columns discretized into n=4 equi-height bins and dummy-coded to avoid multicollinearity. For Matryoshka, we use k=20 joinable tables per query and a correlation-based pruning threshold of 0.55 for regression and 0.01 for classification. Finally, we train a Random Forest and Generalized Linear Models (Gamma and Logistic regression) as our downstream models. We motivate our choice by having one non-linear estimator and a linear one with a non-Gaussian assumption.

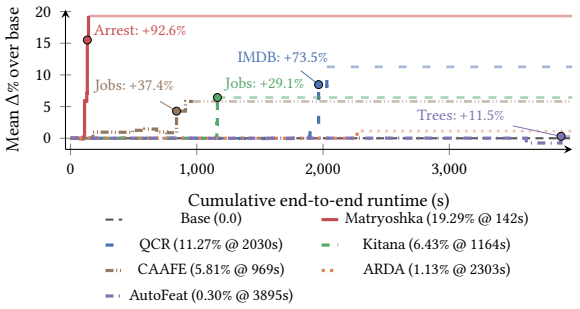
Table 3: Average Relative Improvement (%) over the base model across feature-engineering setups. GR/LR: Gamma (7 regression tasks) and Logistic (4 classification tasks) GLM models.

	Join Key Only			Categ. Features			Binned Features		
Method	RF	GLM (GR)	GLM (LR)	RF	GLM (GR)	GLM (LR)	RF	GLM (GR)	GLM (LR)
ARDA	+8.0	+0.0	**+0.0**	+1.1	+0.0	**+0.0**	+17.3	+0.0	+0.8
AutoFeat	+24.7	+0.2	-3.6	+0.3	**+0.2**	-2.7	+9.0	+0.0	+12.0
CAAFE	+12.5	**+13.6**	-12.5	+5.8	-28.1	-4.1	+0.1	-5.1	+0.0
Kitana	+0.6	+0.0	**+0.0**	+6.4	-27.9	-7.4	+1.3	+0.0	+0.0
QCR	+21.2	+0.2	-6.1	+11.3	-30.6	**+0.0**	+3.5	**+0.6**	**+22.6**
Matryoshka	**+36.1**	+0.0	-10.0	**+19.3**	-33.1	-9.0	**+17.9**	-0.2	+19.5

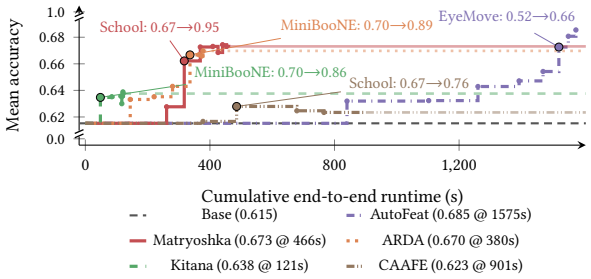
End-to-end Results: Table 3 reports the average relative improvement over the base model under a random forest (RF) and a GLM across the three feature setups. Under RF, Matryoshka achieves the highest gains in every setup (+36.1%, +19.3%, +17.9%), and the gains grow as less query information is available. The GLM column is weaker and uneven for all methods: its score degradations concentrate on the Gamma regression (GR) under categorical preprocessing, where un-binned numeric lake features destabilize the exponential link on long-tailed targets, while logistic regression (LR) turns strongly positive once numeric features are binned.

Anytime Augmentation: We further study the speed of augmentation on the ITW Categorical and AutoFeat benchmarks, reporting aggregate quality as a function of time in analogy to AutoML. We score augmented tables on ITW Categorical with our Random Forest pipeline, reporting mean Δ% metrics (RMSE for regression, F1 for classification) over the base table, and on AutoFeat with the original ensemble of Random Forest, Gradient Boosting, Extra Trees, and XGBoost, reporting mean accuracy. A target-leakage safeguard removes numeric columns with Pearson correlation > 0.99 or OLS fit $R^2 > 0.999$ to the target. On ITW Categorical (Figure 7 (a)), Matryoshka reaches +19.29% after 142 s and is the only method that delivers robust improvements: QCR matches the quality (+11.3%) but supports only regression and is 14× slower, ARDA and AutoFeat reject nearly all candidates because their coreset sampling discards join-relevant rows, and CAAFE and Kitana trail by more than 12% at 7–8× the runtime. The AutoFeat benchmark (Figure 7 (b)) favors ARDA and AutoFeat because synthetic join keys yield perfect overlap, which mitigates the sampling issues. Matryoshka still attains higher mean accuracy than ARDA despite its linear proxy, which identifies features whose predictive value transfers to non-linear models, and comes within 0.02 of AutoFeat's best accuracy at 3.4× lower runtime. Kitana and CAAFE are not competitive.

Impact of Selected Features: On Real Estate, Matryoshka's forward selection retrieves features on permit history, the local



(a) ITW Categorical: 11 query tables, Δ% vs base.



(b) AutoFeat lakes: 8 query tables, mean accuracy.

Figure 7: Anytime Augmentation: (a) ITW Categorical with the RMSE/F1 Δ% metric. (b) The eight AutoFeat lakes with mean accuracy. In both panels, each method folds in its datasets from slowest to fastest by its measured end-to-end runtime.

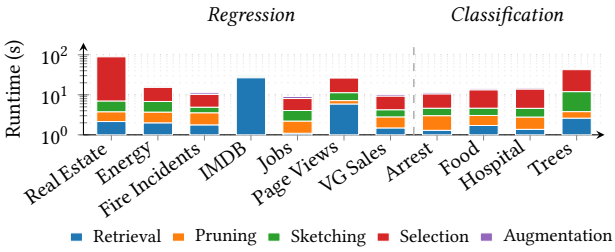


Figure 8: Per-step Runtime Breakdown.

zoning regime, and nearby fire-suppression systems, signals of renovation activity, lot density, and the code compliance of surrounding buildings that plausibly drive real estate value. In contrast, QCR's correlation-only retrieval returns 15 duplicate features and opaque operational identifiers that spuriously correlate with price, degrading the RMSE by +1.0%.

Matryoshka Runtime Breakdown: Figure 8 decomposes the end-to-end runtime into five online steps: *Retrieval* issues the overlap query *OQ* for the top-k joinable tables, *Pruning* filters low-signal candidates, *Sketching* assembles the Gram matrix sketches via the feature space query *FQ*, *Selection* runs greedy forward selection with the F-GS proxy, and *Augmentation* materializes the join for the chosen features. Selection dominates (68.5% of mean runtime) because every iteration re-evaluates all remaining candidates via incremental F-GS updates, a cost amplified on large post-pruning pools such as Real Estate (1460) and Trees (1370). Retrieval accounts for 17.3% (4.0 s mean) and scales with the lake size, from 26.5 s for IMDB on Git (~1M tables) to below 1 s on the smaller CUK lake. Sketching contributes 11.0% (2.6 s), and Pruning and Augmentation together stay near 3%.

Trading Offline Cost for Online Speed: Matryoshka deliberately shifts cost from repeated online querying to a one-time offline sketch precomputation. Table 4 reports this cost: the sketches occupy 89 GiB on NYC, 2.34 TiB on CUK, and 1.32 TiB on Gittables, with indexing times of 5 to 107 hours capturing pairwise cofactors of every candidate column. The investment pays off online: Matryoshka augments a query against the largest lake (Gittables, ~1M tables) within seconds to ~30 s (Figure 8) without reading or joining the raw lake tables at query time.

7.3 Leave-One-Out Benchmark

We stress-test robustness at lake scale with a leave-one-out benchmark on the Kitana [10] lake, as suggested by the authors.

Setup: The lake contains 258 annotated NYC open-data tables with continuous targets and join keys. Each table in turn serves as the query, is removed from the index, augmented with the remaining candidates, and scored with a 5-fold cross-validated Random Forest. Two queries with corrupted annotated targets fail to train the base model and are dropped, leaving 256 tables.

Results: Table 5 reports per-method win ($\Delta > 0$), tie, and loss rates, the engaged summary of Δ% RMSE over the no-augmentation baseline, augmentation time, and worker memory. Matryoshka wins on 73.4% of queries against Kitana's 24.2%, with an engaged median of +22.4% and p10 of −2.1% versus Kitana's +6.9% and −45.0%. ARDA wins on 36.7% but its engaged mean of −115% reflects spurious correlations, and AutoFeat and CAAFE do not augment on 75.4% and 64.8% of queries. QCR wins on 78.5% with an engaged median of +21.0% in 25.5 min at 0.4 GB peak RAM, as the small Kitana tables keep its join-and-test materialization inexpensive. Matryoshka matches QCR on quality while causing less harm, with the best p10 (−2.1%) and loss rate (12.9%), at 16.7 GB peak memory due to a permissive pruning threshold of 0.1.

7.4 Comparison with General Discovery Systems

Figure 9 compares Matryoshka with general-purpose join discovery systems at a retrieval size of k=50. General join discovery returns all retrieved features and defers costly selection to a high-dimensional table, where the model may overfit or fail to converge when #rows≤#features. Every retrieval engine (DeepJoin, JOSIE,

Table 4: Offline Storage and Index Build Runtime.

	NYC		CUK		Gittables	
System	*Storage*	*Build*	*Storage*	*Build*	*Storage*	*Build*
Matryoshka	89 GiB	4 h 54 m	2.34 TiB	95 h 24 m	1.32 TiB	106 h 48 m
QCR [34]	1.4 GiB	11 m	22 GiB	6 h 1 m	92 GiB	33 h 8 m
LSH Ensemble [47]	32 MiB	1 h 48 m	239 MiB	3 h 7 m	13 GiB	1 h 52 m
JOSIE [46]	22 GiB	58 m	52 GiB	2 h 00 m	32 m —	
DeepJoin [15]	101 MiB	11 m	1.3 GiB	12 m	72 GiB	5 h 50 m

Table 5: Leave-one-out Benchmark on Kitana (n=256 tables).

	Rates			Engaged (Δ%$RMSE \neq 0$)			Time	RAM	
	W %	T %	L %	p10	p50	mean	min	p50, MB	max, GB
ARDA	36.7	41.8	21.5	−174.0	21.7	−115.1	83.7	44.7	1.1
AutoFeat	12.1	75.4	**12.5**	−98.4	−0.7	0.7	180.8	7.4	11.0
Kitana	24.2	53.1	22.7	−45.0	6.9	4.7	31.6	**0.9**	1.4
QCR	**78.5**	**5.1**	16.4	−9.6	21.0	26.5	**25.5**	4.3	**0.4**
CAAFE	17.2	64.8	18.0	−95.6	−1.5	−25.6	85.8	6.8	1.3
MATRYOSHKA	73.4	13.7	12.9	**−2.1**	**22.4**	**27.0**	132.3	6.7	16.7

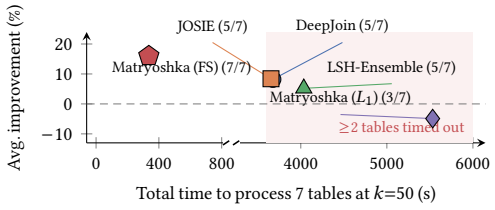
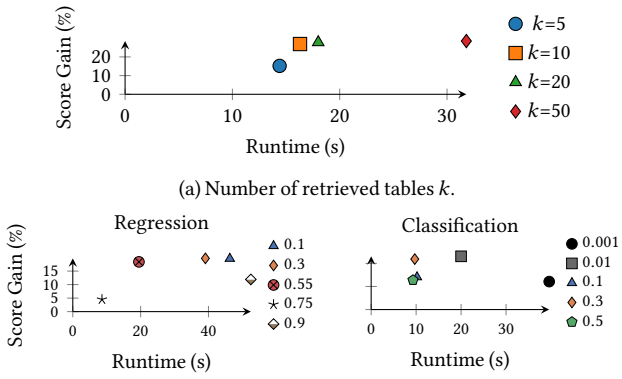


Figure 9: Comparison with General Join Discovery Methods.



(a) Number of retrieved tables k.

(b) Correlation-pruning threshold τ.

Figure 10: Mean Score Gain vs. Runtime across ITW Queries.

LSH-Ensemble) paired with a generic L_1 selector exceeds the per-table 1,800 s budget, augmenting only 3-5 of 7 queries, whereas MATRYOSHKA (*FS*) augments all seven in ~336 s with the best improvement. Yet, MATRYOSHKA in retrieval-only mode (L_1) degrades the downstream score by −4.9%.

7.5 Micro-Benchmarks

We conduct micro-benchmarks to evaluate individual components of MATRYOSHKA under varying assumptions.

Impact of Exploration Budget: We evaluate how the number of top-k joinable tables of the overlap query OQ affects downstream prediction and runtime. Figure 10 (a) varies $k \in \{5, 10, 20, 50\}$, averaged across all 11 ITW query tables. Increasing k from 5 to 20 steadily improves the gain from 15.2% to 27.8% at moderate runtime growth, whereas k=50 yields only a marginal gain (28.5%) while nearly doubling the runtime to 31.8 s.

Figure 11: Fwd Runtime of Full QR vs. Incremental F-GS.

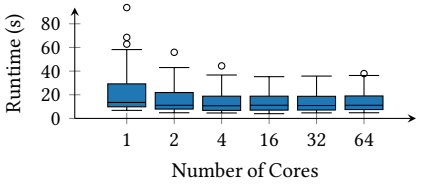


Figure 12: End-to-end Runtime with Degree of Parallelism.

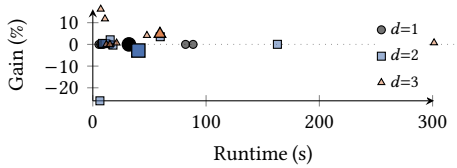
Impact of Pruning Threshold: Figure 10 (b) shows how the pruning threshold τ affects score gain and runtime across all 11 ITW query tables, approximating Pearson correlation via QCR for regression and using the correlation ratio for classification. Moderate thresholds achieve the highest regression gain of roughly 20%: raising τ to 0.75 eliminates over 99% of candidates on Energy (2,918 down to 25) and cuts runtime from 26.5 s to 11.3 s, but the gain drops to 4.5%, and on Real Estate only 4 of 2,992 features remain and the gain turns negative. Classification pruning is more aggressive because the correlation ratio applies per class: on Trees, $\tau \geq 0.3$ prunes all but 3 of 3,024 features and eliminates most of the gain, while on Arrest, τ=0.1 retains 112 of 1,083 features yet achieves 98% of the F1 gain.

Incremental Updates: We evaluate the runtime benefit of incremental F-GS factorization single-threaded, so parallelization does not hide effects. Figure 11 reports the runtime and number of selected features for the 7 ITW regression query tables. The two approaches are nearly identical on most datasets because few features are selected, and the cubic-vs.-quadratic gap becomes visible only for large selected sets, as on Energy with 265 candidates.

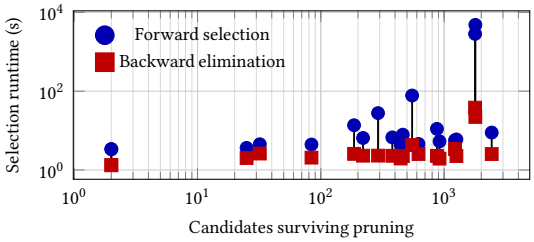
Impact of Parallelization: We further evaluate the speedup of parallelizing the discovery pipeline (during sketching and feature selection). Figure 12 shows the end-to-end runtime over all 11 ITW query tables with increasing degree of parallelism from 1 to 64. The median runtime drops from 13.5 s at 1 core to 10.7 s at 4 cores, a 1.26× speedup. Notably, the upper quartile dominated by large candidate

Table 6: Computation/Ray Overhead (OH) Fwd Breakdown.

	Energy			*Real Estate*		
Cores	**Comp. (s)**	**OH (s)**	**OH%**	**Comp.(s)**	**OH (s)**	**OH%**
1	2,896	775	21	2,554	721	22
4	929	728	44	786	564	42
16	291	323	53	251	291	54
32	185	309	63	163	278	63
64	126	322	72	113	289	72



(a) Per-dataset RMSE gain of polynomial features.



(b) Runtime of Fwd/Bwd feature selection.

Figure 13: Polynomial Features and Runtime.

pools improves from 29.2 s to 18.8 s (1.55×), and the heaviest outlier shrinks from 93.7 s to 44.4 s (2.1×). Gains plateau beyond 16 cores because of parallelization overheads. Table 6 quantifies this saturation on the two costliest datasets, Energy and Real Estate (1,318 and 1,416 features), separating *computation time*, dominated by QR factorization with cofactor sum-multiply joins and per-candidate proxy evaluations, from *Ray overhead (OH)*, which covers task dispatch, cofactor block serialization, and inter-worker transfers. At 1 core, OH is 21–22% of the total selection time; scaling to 16 cores, computation drops roughly 10× while OH stays flat, raising its share.

Index Maintenance: We measure two index update scenarios on the CUK lake, for which a ground truth set of unionable pairs exists [14]: S_1 delivers a batch Q of new rows to an already indexed candidate C, and S_2 admits a previously unseen table. Within S_1, *Append-only* mode applies Q against the maintained per-key state of Section 4.4 and rewrites the index entries (*table_index*,*key_col_index*) without re-reading C, whereas *Rebase* mode retrieves the table, materializes $C \cup Q$ in memory, and re-inserts it. On 50 CUK tables ($n_{\text{rows}} \in [46\,K, 740\,K]$, $n_{\text{cols}} \leq 30$), Append-only reaches a median of 5.6 s versus 38.7 s for Rebase, a 6.9× speedup, and S_2 lands at 40.0 s. The state table required by Append-only adds 674 GiB for CUK on top of the 2.34 TiB sketch index.

Non-Linear, Polynomial Features: We capture simple non-linearities by augmenting candidates x with polynomial groups $x^2,\ldots,x^d$. Figure 13 (a) plots mean RMSE gain over the baseline (d=1) and runtime for the 7 ITW regression query tables. Cubic expansion (d=3) yields meaningful gains on datasets with non-linear target interactions, such as VG Sales (+11.8%), Jobs (+16.2%), and Pageviews (+4.1%), while gains stay below 2.5% on saturated signals (Energy, Fire, Real Estate, and IMDB). On Jobs, quadratic terms (d=2, −26.0%) introduce spurious correlations that cubic terms resolve, so degrees beyond 2 must be tuned per dataset to avoid overfitting. Selection time grows from 15.4 s at d=1 to 50.0 s at d=3, which wins on 6 of 7 tables with an average gain of +4.8%.

Feature Selection Algorithms: We compare forward selection (Fwd) and backward elimination (Bwd) on the 7 ITW regression query tables across pruning thresholds $\tau \in \{0.3, 0.5, 0.7\}$. Figure 13 (b) plots the runtime per dataset and threshold, ordered by the number of features after pruning. Bwd is consistently faster, with a mean speedup of up to 217×.However, in mid-sized pools Bwd retains 10 times more features than Fwd, and the noisy columns degrade RMSE (IMDB at τ=0.5: +36.1%; Jobs at τ=0.5: +22.2%; VG Sales at τ=0.7:+12.8%), a practical runtime-accuracy trade-off between the two approaches.

8 RELATED WORK

General Data Discovery: There is a large body of work for general-purpose data discovery aiming at maximizing the number of retrieved joinable or unionable tables [8, 19, 21, 46]. Further, there are systems that aim at holistic assembly of such operations on a unified index and with a unified discovery language [6, 20]. There is more recent work on discovery in distributed setups [3]. The aforementioned systems serve as base operators for ML targeting dataset search. Our solution reuses ideas from Blend [20].

Feature Selection with Custom Indexes: QCR [34], MI index [35], and COCOA [18] leverage bivariate criteria for feature selection: Quadrant Count Ratio, Mutual Information criterion, and Spearman or Pearson correlation. While the index of these systems allows for efficient search of relevant features, the proxy criteria provide rather weak approximations of relevance. These methods evaluate features independently against the target variable, which contradicts the multivariate nature of ML tasks. While some features might be weakly correlated with the target independently, they might possess a high predictive power in combination with others. Moreover, correlation and mutual information do not account for feature redundancy.

Decoupled Feature Selection: Kitana [25] and ARDA [7] implement multivariate feature selection proxies by leveraging ML models: Linear Regression, Random Forest, and Sparse Regression. Here, feature selection considers feature interactions, improving robustness. Metam [22] computes profiles of candidate datasets and applies Reinforcement Learning to iteratively find the optimal subset based on downstream model performance. AutoFeat [27] is a transitive feature discovery method with multi-hop join paths, and ranking based on information-theoretic metrics. CAAFE [24] instead prompts a large language model to iteratively generate and retain feature engineering code on a single joined table.

We provide an index structure and a retrieval strategy that enables multi-variate feature selection. Our approach serves multiple ML tasks and introduces new optimizations for applying linear proxies.

9 CONCLUSIONS

We introduced Matryoshka, an ML-centric data discovery system that unifies joinability search and multivariate feature selection within a single index of offline Gram matrix sketches, which suffice for factorized proxy models and eliminate join materialization, enabling statistically sound "what-if" feature evaluation at lake scale. Correlation-based pruning and multicollinearity-aware greedy selection cut selection cost from cubic to quadratic in the number of selected features. Matryoshka consistently improves predictive scores at competitive discovery cost across diverse lakes and downstream tasks. Future work includes non-linear classification sketches, semantic join keys, and multi-hop discovery.

REFERENCES

[1] Ziawasch Abedjan, John Morcos, Michael N. Gubanov, Ihab F. Ilyas, Michael Stonebraker, Paolo Papotti, and Mourad Ouzzani. 2015. Dataxformer: Leveraging the Web for Semantic Transformations. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper31.pdf Last accessed: 2026-07-13.

[2] AWS. 2022. Registry of Open Data on AWS. https://registry.opendata.aws/. Last accessed: March 22, 2025.

[3] Lennart Behme, Sainyam Galhotra, Kaustubh Beedkar, and Volker Markl. 2024. Fainder: A Fast and Accurate Index for Distribution-Aware Dataset Search. *Proceedings of the VLDB Endowment (PVLDB)* 17, 11 (July 2024), 3269–3282. https://doi.org/10.14778/3681954.3681999

[4] Dan Brickley, Matthew Burgess, and Natasha Noy. 2019. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. In *Proceedings of the International World Wide Web Conference (WWW)* (San Francisco, CA, USA). Association for Computing Machinery, New York, NY, USA, 1365–1375. https://doi.org/10.1145/3308558.3313685

[5] Sonia Castelo, Rémi Rampin, Aécio Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. 2021. Auctus: a dataset search engine for data discovery and augmentation. *Proceedings of the VLDB Endowment (PVLDB)* 14, 12 (July 2021), 2791–2794. https://doi.org/10.14778/3476311.3476346

[6] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[7] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: automatic relational data augmentation for machine learning. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (May 2020), 1373–1387. https://doi.org/10.14778/3397230.3397235

[8] Martin Pekár Christensen, Aristotelis Leventidis, Matteo Lissandrini, Laura Di Rocco, Renée J. Miller, and Katja Hose. 2025. Fantastic Tables and Where to Find Them: Table Search in Semantic Data Lakes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2 ed.) *(Advances in Database Technology)*, Vol. 28. OpenProceedings, 397–410. https://doi.org/10.48786/edbt.2025.32 28th International Conference on Extending Database Technology (EDBT); Conference date: 25-03-2025 Through 28-03-2025. Last accessed: 2026-07-13.

[9] City of New York. 2017. NYC Municipal Building Energy Benchmarking Results. https://www.kaggle.com/datasets/new-york-city/nyc-municipal-building-energy-benchmarking-results. Kaggle. Last accessed: 2026-07-13.

[10] Columbia University Database Group. 2024. Kitana Experiment Benchmark. https://github.com/zachary62/Kitana-Experiment. Last accessed: 2026-07-13.

[11] Columbia University Database Group. 2024. Kitana Repository. https://github.com/cudbg/Kitana. Last accessed: 2026-07-13.

[12] Peter Craven and Grace Wahba. 1979. Smoothing Noisy Data with Spline Functions: Estimating the Correct Degree of Smoothing by the Method of Generalized Cross-Validation. *Numer. Math.* 31, 4 (1979), 377–403. https://doi.org/10.1007/BF01404567

[13] DELFT Data Management Group. 2024. AutoFeat Repository. https://github.com/delftdata/autofeat. Last accessed: 2026-07-13.

[14] Yuhao Deng, Chengliang Chai, Lei Cao, Qin Yuan, Siyuan Chen, Yanrui Yu, Zhaoze Sun, Junyi Wang, Jiajun Li, Ziqi Cao, Kaisen Jin, Chi Zhang, Yuqing Jiang, Yuanfang Zhang, Yuping Wang, Ye Yuan, Guoren Wang, and Nan Tang. 2024. LakeBench: A Benchmark for Discovering Joinable and Unionable Tables in Data Lakes. *Proceedings of the VLDB Endowment (PVLDB)* 17, 8 (2024), 1925–1938. https://doi.org/10.14778/3659437.3659448 Last accessed: 2026-07-13.

[15] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2023. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. *Proceedings of the VLDB Endowment (PVLDB)* 16, 10 (2023), 2458–2470. https://doi.org/10.14778/3603581.3603587 Last accessed: 2026-07-13.

[16] Ted Dunning. 2021. The t -digest: Efficient estimates of distributions. *Softw. Impacts* 7, 100049 (Feb. 2021), 100049.

[17] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. In *AutoML@ICML Workshop*. https://arxiv.org/abs/2003.06505

[18] M. Esmailoghli, J. Quiane-Ruiz, and Z. Abedjan. 2021. COCOA: Correlation Coefficient-Aware Data Augmentation. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra (Eds.). OpenProceedings.org, Nicosia, Cyprus, 331–336. https://doi.org/10.5441/002/edbt.2021.30

[19] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2022. MATE: Multi-Attribute Table Extraction. *Proceedings of the VLDB Endowment (PVLDB)* 15, 8 (2022), 1684–1696. https://www.vldb.org/pvldb/vol15/p1684-esmailoghli.pdf Last accessed: 2026-07-13.

[20] Mahdi Esmailoghli, Christoph Schnell, Renée J. Miller, and Ziawasch Abedjan. 2024. Blend: A Unified Data Discovery System. arXiv:2310.02656 [cs.DB] https://arxiv.org/abs/2310.02656

[21] Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, and Renée J. Miller. 2023. Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning. *Proceedings of the VLDB Endowment (PVLDB)* 16, 7 (2023), 1726–1739. https://doi.org/10.14778/3587136.3587146 Last accessed: 2026-07-13.

[22] Sainyam Galhotra, Yue Gong, and Raul Castro Fernandez. 2023. METAM: Goal-Oriented Data Discovery. arXiv:2304.09068 [cs.DB] https://arxiv.org/abs/2304.09068

[23] Noah Hollmann. 2024. CAAFE Repository. https://github.com/noahho/CAAFE. Last accessed: 2026-07-13.

[24] Noah Hollmann, Samuel Müller, and Frank Hutter. 2023. Large Language Models for Automated Data Science: Introducing CAAFE for Context-Aware Automated Feature Engineering. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/8c2df4c35cdbee764ebb9e9d0acd5197-Abstract-Conference.html Last accessed: 2026-07-13.

[25] Zezhou Huang, Pranav Subramaniam, Raul Castro Fernandez, and Eugene Wu. 2023. Kitana: Efficient Data Augmentation Search for AutoML. arXiv:2305.10419 [cs.DB] https://arxiv.org/abs/2305.10419

[26] Madelon Hulsebos, Çagatay Demiralp, and Paul Groth. 2023. GitTables: A Large-Scale Corpus of Relational Tables. *Proc. ACM Manag. Data* 1, 1 (2023), 30:1–30:17. https://doi.org/10.1145/3588710 Last accessed: 2026-07-13.

[27] Andra Ionescu, Kiril Vasilev, Florena Buse, Rihan Hai, and Asterios Katsifodimos. 2024. AutoFeat: Transitive Feature Discovery over Join Paths. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1861–1873. https://doi.org/10.1109/ICDE60146.2024.00150

[28] Kaggle. 2025. Kaggle Datasets. https://www.kaggle.com/datasets. Last accessed: March 22, 2025.

[29] P. C. Mahalanobis. 2018. ON THE GENERALIZED DISTANCE IN STATISTICS. *Sankhyā: The Indian Journal of Statistics, Series A (2008-)* 80 (2018), pp. S1–S7. https://www.jstor.org/stable/48723335

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 561–577.

[31] Tiago Nunes. 2023. IMDB Top 5000 Movies. https://www.kaggle.com/datasets/tiagoadrianunes/imdb-top-5000-movies. Kaggle. Last accessed: 2026-07-13.

[32] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[33] Patricia Ryser-Welch. 2023. United Kingdom Real Estate and Population Data. https://www.kaggle.com/datasets/patriciaryserwelch/united-kingdom-real-estate-and-population-data. Kaggle. Last accessed: 2026-07-13.

[34] Aécio Santos, Aline Bessa, Christopher Musco, and Juliana Freire. 2022. A Sketch-based Index for Correlated Dataset Search. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 2928–2941. https://doi.org/10.1109/ICDE53745.2022.00264

[35] Aécio Santos, Flip Korn, and Juliana Freire. 2024. Efficiently Estimating Mutual Information Between Attributes Across Tables. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 193–206. https://doi.org/10.1109/icde60146.2024.00022 Last accessed: 2026-07-13.

[36] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 3–18. https://doi.org/10.1145/2882903.2882939 Last accessed: 2026-07-13.

[37] Gregory Smith. 2016. Video Game Sales — kaggle.com. https://www.kaggle.com/datasets/gregorut/videogamesales. [Accessed 21-10-2025].

[38] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.* 13, 2 (2011), 22–30. https://doi.org/10.1109/MCSE.2011.37 Last accessed: 2026-07-13.

[39] Bas A. M. van Geffen. 2018. *QR Decomposition of Normalised Relational Data*. Master's thesis. University of Oxford, Oxford, UK. https://www.cs.ox.ac.uk/dan.olteanu/theses/Bas.vanGeffen.pdf Master of Science in Computer Science, Kellogg College; Supervisor: Prof. Dan Olteanu. Last accessed: 2026-07-13.

[40] Ritchie Vink. 2024. Polars: Blazingly Fast DataFrames in Rust and Python. https://github.com/pola-rs/polars. Last accessed: 2026-07-13.

[41] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[42] Yair Wand and Richard Y. Wang. 1996. Anchoring data quality dimensions in ontological foundations. *Commun. ACM* 39, 11 (Nov. 1996), 86–95. https://doi.org/10.1145/240455.240479

[43] Richard Y. Wang and Diane M. Strong. 1996. Beyond accuracy: what data quality means to data consumers. *J. Manage. Inf. Syst.* 12, 4 (March 1996), 5–33. https://doi.org/10.1080/07421222.1996.11518099

[44] Xiaoying Wang, Weiyuan Wu, et al. 2024. ConnectorX: Accelerating Data Loading From Databases to Dataframes. https://github.com/sfu-db/connector-x. Last accessed: 2026-07-13.

[45] Amy Zhao Yu, Shahar Ronen, Kevin Hu, Tiffany Lu, and César A. Hidalgo. 2016. Pantheon 1.0, a manually verified dataset of globally famous biographies. *Scientific Data* 3, 1 (Jan. 2016). https://doi.org/10.1038/sdata.2015.75 Last accessed: 2026-07-13.

[46] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 847–864. https://doi.org/10.1145/3299869.3300065 Last accessed: 2026-07-13.

[47] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proceedings of the VLDB Endowment (PVLDB)* 9, 12 (2016), 1185–1196. https://doi.org/10.14778/2994509.2994534 Last accessed: 2026-07-13.