Coresets over Multiple Tables for Feature-rich and Data-efficient Machine Learning

Jiayi Wang
Tsinghua University

Chengliang Chai
Tsinghua University

Nan Tang
QCRI

Jiabin Liu
Tsinghua University

Guoliang Li
Tsinghua University

ABSTRACT

Successful machine learning (ML) needs to learn from good data. However, one common issue about train data for ML practitioners is the lack of good features. To mitigate this problem, feature augmentation is often employed by joining with (or enriching features from) multiple tables, so as to become feature-rich ML. A consequent problem is that the enriched train data may contain too many tuples, especially if the feature augmentation is obtained through 1 (or many)-to-many or fuzzy joins. Training an ML model with a very large train dataset is data-inefficient. Coreset is often used to achieve data-efficient ML training, which selects a small subset of train data that can theoretically and practically perform similarly as using the full dataset. However, coreset selection over a large train dataset is also known to be time-consuming.

In this paper, we aim at achieving both feature-rich ML through feature augmentation and data-efficient ML through coreset selection. In order to avoid time-consuming coreset selection over a feature augmented (or fully materialized) table, we propose to efficiently select the coreset without materializing the augmented table. Note that coreset selection typically uses weighted gradients of the subset to approximate the full gradient of the entire train dataset. Our key idea is that the gradient computation for coreset selection of the augmented table can be pushed down to partial feature similarity of tuples within each individual table, without join materialization. These partial feature similarity values can be aggregated to estimate the gradient of the augmented table, which is upper bounded with provable theoretical guarantees. Extensive experiments show that our method can improve the efficiency by nearly 2 orders of magnitudes, while keeping almost the same accuracy as training with the fully augmented train data.

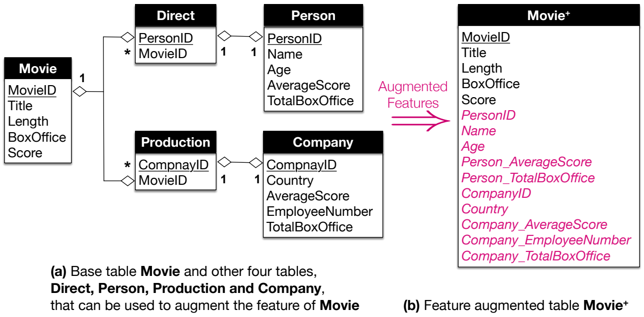
1 INTRODUCTION

Feature-rich machine learning (ML) [8, 9, 25, 29] means that ML models are trained with enough and good features. Given train data, *data-efficient ML* [2, 3, 32] aims at training ML models faster without sacrificing the model accuracy. Putting them together, the goal is to efficiently train robust ML models.

For achieving **feature-rich** ML, the widely used practice is to enrich features by joining a base table with multiple tables, *a.k.a. feature augmentation* [9, 26].

Example 1. *[Feature-rich ML through Feature Augmentation.] Consider an ML task that predicts the* `Score` *value of a movie based on attributes* (i.e., *features)* `MovieID, Title, Length` *and* `BoxOffice`*, as shown in the* `Movie` *table in Figure 1(a). Intuitively, because many important features are missing, such as the features of directors and the actors of a movie, it is hard to train a good ML model.*

Consider four tables `Direct`, `Person`, `Production` *and* `Company`*, which can be joined, directly or indirectly, with the* `Movie` *table through predefined joins, as shown in Figure 1(a). The primary key of each table is annotated with an underline* (e.g., `MovieID` *for table* `Movie`*). These tables can be joined with either 1-to-1 relationship or 1-to-many* (i.e., *1-to-∗) relationship. The feature-rich table with new features augmented through joins, denoted by* `Movie`$^+$*, is shown in Figure 1(b).*



(a) Base table **Movie** and other four tables, **Direct, Person, Production and Company**, that can be used to augment the feature of **Movie**

(b) Feature augmented table **Movie**$^+$

Figure 1: Feature-rich ML through feature augmentation.

For **data-efficient** ML, besides traditional methods (*e.g.*, stochastic gradient descent and its variants), there are many recent efforts on selecting a train data subset that can theoretically and practically perform on par with the full dataset, *a.k.a. coresets* [13, 34].

In order to achieve both **feature-rich** and **data-efficient** ML, an intuitive solution is to first conduct feature augmentation through joins across multiple relational tables, followed by performing coreset selection over the train data with enriched features. In Figure 2, this strategy is depicted by first following the "⇒" arrow from the base table T and then the "⟶" arrow.

Before we discuss the problem we study in this paper, let's analyze the benefits and limitations of the aforementioned strategies, using real experiments (see more details in § 6).

Example 2. *[Benefits and Limitations of Existing Strategies.] Please refer to Figure 3 for the following discussions.*

Train with the Base Table. *If we train an ML model using the base table T for a multi-classification task, we can achieve an accuracy* 0.61 *(Figure 3(a)–❶) and use 11 minutes for training (Figure 3(b)–❶). We consider this as a baseline.*

Train with the Coreset. *If we first compute the coreset $C(T)$ of the base table T, and then train an ML model using the computed coreset, we can also achieve an accuracy* 0.61 *(Figure 3(a)–❷) but use 2 minutes in total for selecting the coreset and training with the coreset (Figure 3(b)–❷). This shows that using coreset can significantly reduce the training time without sacrificing the accuracy.*

Train with the Augmented Table. *As we know, the process of feature augmentation consists of 1 (many)-to-many joins [8, 9, 29, 41]. In this situation, the size of the augmented table T^+ is likely to be much larger than the base table T. Although training with the augmented table*

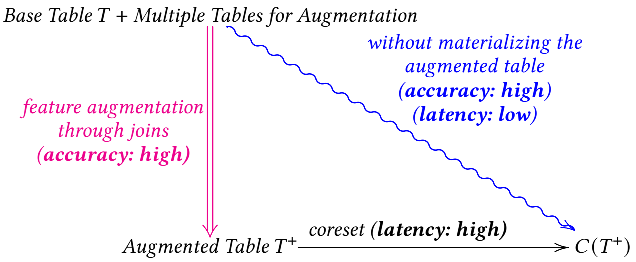
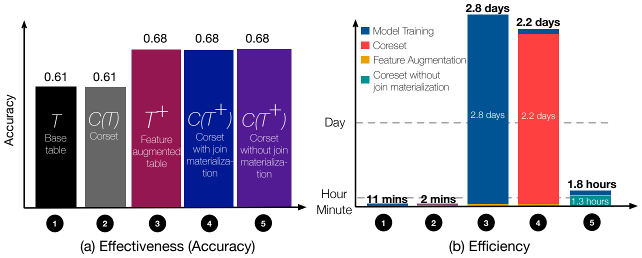

Base Table T + Multiple Tables for Augmentation
without materializing the augmented table
(accuracy: high)
(latency: low)
feature augmentation through joins
(accuracy: high)
coreset (latency: high)
Augmented Table T^+
$C(T^+)$

Figure 2: Various design choices.



(a) Effectiveness (Accuracy)

(b) Efficiency

Figure 3: Comparison of various design choices.

can achieve a much higher accuracy 0.68 *(Figure 3(a)–❸), it takes around 2.8 days for training over 10 million tuples (Figure 3(b)–❸).*

Train with the Coreset of the Augmented Table. *If we first compute the coreset of the augmented table, and then train with this coreset, we can achieve the same accuracy* 0.68 *(Figure 3(a)–❹). However, because coreset computation on a large table is time-consuming which takes 2.2 days in this case, putting the time of feature augmentation (5 minutes) and training (0.5 hours) together, it takes around 2.2 days (Figure 3(b)–❹).*

Example 2 tells us that feature augmentation can significantly increase the accuracy and training with a coreset can significantly reduce the time. However, computing the coreset over a large table (*e.g.,* the augmented table) is time-consuming.

In order to efficiently compute the coreset of the augmented table, the **problem** we aim to tackle is whether we can **compute the coreset of the augmented table without join materialization.** This strategy is depicted in Figure 2 by following the "⤳" arrow from the base table.

Key Idea. Our solution is inspired by the classical SQL query optimization technique **pushdown** that moves predicates in the WHERE clause closer to the tables they refer to. Next let's build the connection between pushdown and our problem. Generally speaking, coreset computation is to select a subset of tuples, and use the weighted gradients of these tuples to approximate the full gradient of the entire train data. In our context, **pushdown for coreset over multiple tables** means that we can approximately compute the gradient of each individual table and sum up these gradients from multiple tables to compute the full gradient of the joined table. That is, the *gradient computation* is pushed down.

Challenges. Applying *pushdown* to compute the coreset of the augmented table without join materialization faces two challenges.

- **(C1)** How to deal with each individual table so that the estimated full gradients can be bounded with theoretical guarantees, and thus the accuracy will not be sacrificed?
- **(C2)** How to efficiently aggregate information distributed in different tables to well approximate the full gradient?

Our Methodology. To address the above challenges, we propose feature-Rich and data-Efficient machine learning with Coreset selection withOut join materializatioN, namely RECON, which groups the tuples in each table based on predefined group key and compute the gradient bounds of these groups based on tuple-wise partial feature similarity. We prove that the full gradient can be bounded using these partial feature similarity values of different groups (for **C1**). Based on the computed partial feature similarity values, we prove that the coreset computation problem over multiple tables is NP-hard and has the submodular property, so we use a greedy algorithm with an approximate ratio to aggregate the partial feature similarity values in different tables, and then compute a well-performed coreset (for **C2**).

Figure 3 shows that RECON can significantly reduce the total time from more than 2 days to 1.8 hours (Figure 3(b)–❺) without sacrificing the accuracy (Figure 3(a)–❺). See § 6 for more details.

Contributions. In this paper, we make a first attempt to study coreset selection over multiple tables without full materialization. In sum, we make the following notable contributions.

(1) We introduce coreset selection over a single table, its sample-based optimization, and our key idea of supporting coreset selection over multiple tables without full materialization in § 3.

(2) We provide theoretical guarantees on the gradient approximation of the augmented table using partial feature similarity values computed from each single table. We also prove that the coreset computation over multiple tables is an NP-hard problem with the submodular property. Putting them together, we theoretically show that gradient approximation error of coreset for the augmented table can be upper bounded by the computation of each individual table, without physically joining them in § 4.

(3) We propose an efficient greedy algorithm to compute the coreset of the augmented table without materializing the joins. To be specific, we utilize a dynamic programming algorithm to efficiently aggregate partial feature similarity values from multiple tables and finally bound the full gradient in § 5.

(4) We conducted extensive experiments on 5 real world datasets to evaluate the efficiency of our proposed method. The experimental results demonstrate that our method can improve the efficiency to nearly 2 orders of magnitudes by training over the selected coreset ($\sim$ 0.1% of the entire train data), while keeping almost the same accuracy as training with the fully augmented train data in § 6.

2 RELATED WORK

Generally speaking, there are two ways of table enrichment for better training ML models, either by adding more columns (*i.e.,* feature augmentation) or more tuples (*i.e.,* data acquisition [40]). Both are common in practice and they are complementary to each other. Our proposal falls into the category of feature augmentation.

Feature augmentation. A ***brute force*** solution is to execute joins to augment new features. Another line of research focuses on ***avoiding unnecessary joins*** (*e.g.,* Kumar et al. [26] and Shah et al. [42]), when the foreign join key has already contained all the information of the external table. There are also studies [9, 29] on ***iterative feature augmentation***, which iteratively select an optimal subset of tables, such that features augmented from these tables can significantly improve the model performance.

Different from (iterative) feature augmentation, by following the setting in [16, 25, 28], we assume that which joins are useful (*i.e.,* which attributes should be added) are given. Besides, iterative feature augmentation [9, 29] needs to join tables, train over the result and test the performance iteratively, which is rather time-consuming. Therefore, RECON can be leveraged to accelerate this process in each iteration. In terms of avoiding unnecessary joins, we can take it as a filtering of our method. That is, the outputs of the method are the input of our method. Note that [26, 42] do not support to avoid one-to-many, many-to-many and fuzzy joins. In summary, the above methods are complementary to our proposal.

Note that when the number of features is large, **feature selection** (*a.k.a.* variable selection or attribute selection), is the process of selecting a subset of *good* features for use in model construction and has been extensively studied in the ML community [15], which is orthogonal to feature augmentation tackled in this paper.

Coreset selection. Existing coresete selection algorithms are designed for one table. Huang et al. [18] proposed to select and update the coreset while training. The goal is to use the loss of training tuples in the coreset to approximate the overall training loss of the entire dataset. Since they have to train the model, it is rather time-consuming. To address this, works [6, 7] focused on selecting the coreset without training in advance, but they are customized to particular model types respectively. The high level idea is to compute a sensitive score for each tuple. The higher the score, the more likely the tuple should be a member of the coreset. Dataset condensation [45, 46] tries to synthesize a small set of train data, instead of selecting a small set of train data. Hence, the related work is closer to knowledge distillation than coreset selection. Moreover, these papers are only tested on image data. In terms of tabular data, it is not verified whether it can synthesize a small set of train data while well preserving the labels. The other typical line of works [21, 23, 32, 33] focused on selecting the coreset to approximate the full gradient, which is modeled as an optimization problem that can be solved by a framework with three nested loops (see § 3.2).

None of them considers coreset selection over multiple tables. Different from them, we make the first attempt to select coresets over multiple tables without full materialization.

Factorized ML (FML) achieves efficient ML training by decoupling the ML computations through joins to the base tables [8, 20, 24, 25, 28, 36, 39, 41]. The key idea is to reduce redundant linear algebra computations during training over the multi-table joins. Most of these methods only focus on specific ML models or platforms, *e.g.,* Olteanu et al. [36, 41] focus on linear regression models, and [24, 39] are for in-memory databases. Recently, Kumar et al. [8, 20, 25, 28] build a general FML framework by decoupling linear algebra computations from various ML algorithms.

The fundamental difference from us is that they still need to *train over the full join results.* Moreover, we aim at accelerating ML training by reducing the amount of train data through selecting the coreset. However, they focus on the batch gradient descent algorithm for training rather than supporting the stochastic gradient descent, which is widely used in practice due to its high efficiency. In addition, we have also empirically verified that our proposal outperforms FML-based methods for the batch gradient descent scenario (see § 6 for more details).

Algorithm 1: Basic Coreset Algorithm of One Table

```
Input: The train data $T$, coreset size $K$.
Output: A coreset $C \subseteq T$, weight $W = \{w_j\}, |C| = |W| = K$.
$C = \emptyset$;
while $|C| < K$ do
    /*1st loop*/
    for each tuple $t \in T \setminus C$ do
        /*2nd loop*/
        Compute $U(t|C)$ considering all tuples in $T$; /*3rd loop*/
    $t^* = \arg\max_{t \in T \setminus C} U(t|C)$ ;
    $C = C \cup \{t^*\}$;
for $j$ = 1 to $|C|$ do
    $w_j = \sum_{i=1}^{n} \mathbb{I}[j = \arg\min_{c_{j'} \in C} \max_{\theta \in \vartheta} \|\nabla l_i(\theta) - \nabla l_{\gamma(j')}(\theta)\|]$;
return $C, W$;
```

3 CORESET SELECTION FRAMEWORK

3.1 Gradient Descent for Machine Learning

Gradient descent is by far the most popular optimization strategy used in machine learning. Generally speaking, based on a convex and differentiable function, it iteratively tweaks the parameters to minimize a given function to its local minimum.

Let $T = \{t_1, t_2, ..., t_n\}$ be a set of labeled train tuples, where $t_i = (\mathbf{x}_i, \mathbf{y}_i)$, $\mathbf{x}_i \in \mathbb{R}^d$ denotes the feature vector and $\mathbf{y}_i$ is the corresponding label. The objective of training on T is to compute the best parameter θ^* *w.r.t.* an ML model so as to minimize the loss:

$$\theta^* = \arg\min_{\theta \in \vartheta} l(\theta), l(\theta) = \frac{1}{n} \sum_{i=1}^{n} l_i(\theta, t_i) \quad (1)$$

where ϑ denotes the parameter space. For ease of representation, we just use $l_i(\theta)$ to denote the loss of the i-th train example, *i.e.,* $l_i(\theta, t_i)$. Typically, the gradient descent method is always applied to optimize Eq. 1, where the **full gradient**, denoted by $\nabla l(\theta)$, is required to be computed iteratively.

Although some classic incremental gradient methods such as stochastic gradient descent (SGD), can be utilized to accelerate this process, it is still expensive when there are massive train tuples.

3.2 Coreset of One Table T

Coreset. The main problem of learning using a large train dataset T is low efficiency. Hence, instead of learning from entire T, one research direction seeks to answer the question that whether we can compute a small subset $C(T)$ of T such that learning using $C(T)$ can hopefully have the same performance as learning using T. This small subset is called the **coreset** [13, 34]. In the rest of the paper, we will simply write $C(T)$ as C, when it is clear from the context.

To compute the coreset, the SOTA solutions are mainly based on gradient approximation [22, 32]. Intuitively, if θ is the parameter of an ML model trained using the full dataset, and θ' is the parameter

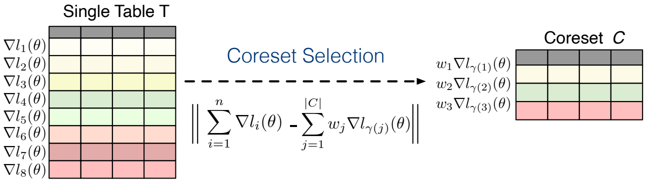

Single Table T
$\nabla l_1(\theta)$
$\nabla l_2(\theta)$
$\nabla l_3(\theta)$
$\nabla l_4(\theta)$
$\nabla l_5(\theta)$
$\nabla l_6(\theta)$
$\nabla l_7(\theta)$
$\nabla l_8(\theta)$
Coreset Selection
$\left\| \sum_{i=1}^{n} \nabla l_i(\theta) - \sum_{j=1}^{|C|} w_j \nabla l_{\gamma(j)}(\theta) \right\|$
Coreset C
$w_1 \nabla l_{\gamma(1)}(\theta)$
$w_2 \nabla l_{\gamma(2)}(\theta)$
$w_3 \nabla l_{\gamma(3)}(\theta)$

Figure 4: Example of coreset selection for a single table.

of the same ML model trained using the subset (or coreset), the goal is $\nabla l(\theta) = \nabla l(\theta')$. Based on gradient approximation, existing solutions can lead to good performance with theoretical guarantees, *i.e.,* $\nabla l(\theta)$ is upper-bounded by $\nabla l(\theta')$. Next let's formally define it.

Coreset selection based on gradient descent. Let $\nabla l(\theta) = \sum_{i=1}^{n} \nabla l_i(\theta)$ be the full gradient training using the entire training dataset, the problem of coreset selection is to minimize the **gradient approximation error** [32] between the full gradient *w.r.t.* T and the weighted sum of gradients *w.r.t.* the coreset C (or coreset gradient).

$$C^* = \underset{C\subseteq T, w_j\geq 0}{\arg\min}\ \max_{\theta\in\vartheta} \underbrace{\| \underbrace{\sum_{i=1}^{n} \nabla l_i(\theta)}_{\text{full gradient}} - \underbrace{\sum_{j=1}^{|C|} w_j \nabla l_{\gamma(j)}(\theta)}_{\text{coreset gradient}} \|}_{\textbf{gradient approximation error}}, \qquad (2)$$

$$s.t.\ |C| \leq K$$

Eq. 2 tries to minimize the gradient approximation error using a coreset of size at most K by considering all possible parameters $\theta \in \vartheta$ (*i.e.,* $\max\limits_{\theta\in\vartheta}$), where "$\|\cdot\|$" denotes the normed difference.

The **full gradient** has been defined earlier as $\nabla l(\theta) = \sum_{i=1}^{n} \nabla l_i(\theta)$. Next, let's focus on explaining how to compute the **coreset gradient** in Eq. 2. We use $\gamma(j) = i, j \in [1, |C|], i \in [1, n]$ to denote that the j-th tuple in C (denoted by c_j) is the i-th tuple in T, *i.e.,* t_i. In other words, γ is an index mapping from C to T. Besides, Eq. 2 potentially contains another important mapping ϕ similar to γ, *i.e.,* $\phi(i) = j, i \in [1, n], j \in [1, |C|]$, which has a close relationship with the weight w. To be specific, let $\phi(i) = j$ denote that we will assign t_i to c_j and use $\nabla l_{\gamma(j)}$ to represent ∇l_i. Each t_i will be assigned to one and only one c_j, but each c_j might be assigned with multiple tuples in T. Based on ϕ, w_j is defined as the weight of the c_j, which is the number of tuples in T assigned to the c_j, *i.e.,* $w_j = |\{t_i | \phi(i) = j, i \in [1, n]\}|$.

Next let's use an example to better illustrate Eq. 2.

Example 3. *Let us consider a special case of the gradients of each tuple, as shown in Figure 4. Suppose that for any* θ, $\nabla l_1(\theta) \approx \nabla l_2(\theta) \approx \nabla l_3(\theta)$, $\nabla l_4(\theta) \approx \nabla l_5(\theta)$ *and* $\nabla l_6(\theta) \approx \nabla l_7(\theta) \approx \nabla l_8(\theta)$. *In this situation, if we want to find an optimal coreset with a size of 3,* i.e., $K = 3$ *based on Eq. 2, the solution can be* $C^* = \{t_1, t_4, t_6\}$ ($\gamma(1) = 1, \gamma(2) = 4$ *and* $\gamma(3) = 6$)*, associated with* $w_1 = 3, w_2 = 2, w_3 = 3$ *because* $\phi(1) = \phi(2) = \phi(3) = 1, \phi(4) = \phi(5) = 2$ *and* $\phi(6) = \phi(7) = \phi(8) = 3$. *In this way,* C^* *will be one of the optimal coresets that can well approximate the full gradient because* $\|\sum_{i=1}^{8} \nabla l_i(\theta) - \sum_{j=1}^{3} w_j \nabla l_{\gamma(j)}(\theta)\| \approx 0$, *which is minimized.*

We can observe from Example 3 that, if $\phi(i) = j$, ∇l_i and $\nabla l_{\gamma(j)}$ are likely to be close such that the gradient approximation error, *i.e.,* Eq. 2, tends to be minimized. Intuitively, computing the coreset is similar to computing the K exemplars [38] of the gradients, if all the gradients of tuples can be computed.

For training with the popular SGD method, the coreset C is first randomly shuffled. Then during each step of gradient descent, suppose that we need to use $c_j \in C$ for gradient update. We first compute the gradient of c_j, say ∇l. Then we use $w_j \nabla l$ to update the parameters of the ML model. The above process is repeated until the ML model converges.

Basic Coreset Algorithm of One Table. The basic coreset algorithm is illustrated in Figure 5(a) and Algorithm 1. Initialized with an empty coreset C, a coreset of size K is achieved using three nested for-loops.

- **1st for-loop (lines 2-8).** Each iteration of the 1st for-loop will add the tuple with the maximum "utility" to the coreset (lines 7-8). The "utility" of a tuple t denotes the reduction of gradient approximation error in Eq. 2 after adding t into the coreset C, denoted by *i.e.,* $U(t|C)$.
- **2nd for-loop (lines 4-6).** Each iteration of the 2nd-iteration will compute the utility of a tuple t that is not in coreset C.
- **3rd for-loop (line 6).** It iterates all tuples in T to compute the utility of tuple t used in the 2nd for-loop.
- **Weights computation (lines 9-10).** It computes the weight of each tuple in the coreset, which will be used to approximate the full gradient.

Apparently, the solution with 3 loops is rather time-consuming. Fortunately, coresets satisfy the submodular property [19] (Theorem 3 in § 4), based on which an efficient method can accelerate the 2nd loop that uniformly samples tuple set $\mathcal{S}$ from $T \setminus C$ and selects the best one from the sampled ones [31]. It holds a $(1 - \frac{1}{e} - \epsilon)$ approximate ratio, where ϵ is related to the sampling ratio.

Coreset by sampling. The optimization of sampling-based coreset selection algorithm is illustrated in Figure 5(b), where the sample $\mathcal{S}$ is a subset of $T \setminus C$ ("\" denotes the operation set difference).

3.3 Coreset of Multiple Tables

As discussed above, to achieve feature-rich ML, the base table has to be augmented to get useful features through joining other tables.

With materialization. A natural solution is first to do feature augmentation by executing the joins, and then use the above coreset selection on the materialized result. Figure 5(c) depicts this solution.

Note that, for feature augmentation, there might have one-to-one, one-to-many, many-to-many and fuzzy joins. Hence, the materialized view might be very large. Consequently, the efficiency of coreset selection is low (see Figure 3(b)-❹).

Without materialization. Our key idea to improve the efficiency of coreset of multiple tables is to estimate the utility of each "group" without join materialization, where a group refers to a set of tuples in the joined results having the same attribute values on a predefined set of attributes, as shown in Figure 5(d). Conceptually, the utility can be estimated by first computing the feature similarity of tuples in each individual table, and then aggregating them using a

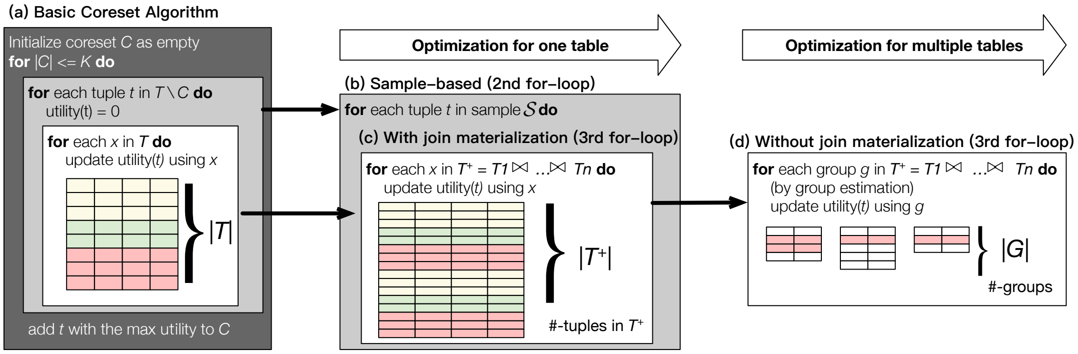

(a) Basic Coreset Algorithm
Initialize coreset C as empty
for |C| <= K do
for each tuple t in T \ C do
utility(t) = 0
for each x in T do
update utility(t) using x
|T|
add t with the max utility to C
Optimization for one table
(b) Sample-based (2nd for-loop)
for each tuple t in sample 𝒮 do
(c) With join materialization (3rd for-loop)
for each x in T⁺ = T1 ⋈ ...⋈ Tn do
update utility(t) using x
|T⁺|
#-tuples in T⁺
Optimization for multiple tables
(d) Without join materialization (3rd for-loop)
for each group g in T⁺ = T1 ⋈ ...⋈ Tn do
(by group estimation)
update utility(t) using g
|G|
#-groups

Figure 5: Basic coreset selection (a), typical optimization techniques (b, c), and our optimization for multiple tables (d).

dynamic programming algorithm without join materialization. By doing so, we can significantly reduce the computation in the 3rd for-loop, thus improving the overall efficiency (see Figure 3(b)-❺).

4 GRADIENT APPROXIMATION ERROR BOUNDED BY GROUPS

In this section, we will build the theoretical bound of gradient computation for coresets over multiple tables. Afterwards, we will introduce the algorithms by following the theoretical bounds (§ 5).

Let T be the base table, $T_1, T_2, \ldots, T_m$ be the tables that can be used to augment the features of T, and T^+ be the feature-augmented table through predefined joins, with $|T^+| = N$. We will prove that the gradient approximation error of a coreset *w.r.t.* T^+ can be upper-bounded using the groups (or partitions) of T^+.

Recall that minimizing Eq. 2 is closely related to the parameter θ. Unfortunately, the entire parameter space ϑ is too expensive to explore. We will first prove that the gradient approximation error is upper-bounded for a fixed parameter θ and given groups (§ 4.1). We will then generalize the above result to the parameter space ϑ for given groups (§ 4.2). We will close this section by discussing the connection between groups in the augmented single table (*i.e.,* T^+) and multiple tables (§ 4.3).

Building upon the above results, later § 5 will describe how to compute groups from multiple tables (*i.e.,* $T, T_1, \ldots, T_m$) and use them to bound the gradient approximation error of the coreset of T^+ without materializing T^+.

<u>Notation.</u> Similar to § 3, we use ∇l_i^+ to denote the loss of the i-th training example in T^+, *i.e.,* $l_i^+(\theta, t_i)$.

4.1 Upper Bound of a Fixed θ and Given Groups

Recap that we have a one-to-one mapping $\phi_\theta : T^+ \to C$ between tuples in T^+ and C. $\phi_\theta(i) = j$ denotes that t_i^+ in T^+ should be assigned to the j-th tuple in the coreset. With this mapping, we get:

$$\sum_{j=1}^{|C|} w_j \nabla l^+_{\gamma(j)}(\theta) = \sum_{i=1}^{N} \nabla l^+_{\gamma(\phi_\theta(i))}(\theta)$$

so Eq. 2 can be rewritten as [32]:

$$\|\sum_{i=1}^{N} \nabla l_i^+(\theta) - \sum_{j=1}^{|C|} w_j \nabla l^+_{\gamma(j)}(\theta)\| = \|\sum_{i=1}^{N}\left(\nabla l_i^+(\theta) - \nabla l^+_{\gamma(\phi_\theta(i))}(\theta)\right)\| \tag{3}$$

Let $\mathcal{A}$ be the predefined attribute set (or the grouping key), based on which tuples in T^+ are divided into a set $\mathcal{G}$ of disjoint groups $g = |\mathcal{G}|$ such that each group $\mathcal{G}_i \in \mathcal{G}$ contains tuples with the same values on $\mathcal{A}$ (see § 6.1 for more implementation details). We then use $G_i, i \in [1, g]$ to denote the set of indexes (corresponding to T^+) of tuples in $\mathcal{G}_i$, *i.e.,* $G_i = \{k | t_k^+ \in T^+, t_k^+ \in \mathcal{G}_i\}$. That is, $\cup_{i=1}^{g} G_i = \{1, 2, \ldots, N\}$. With such grouping, different from [32], the summation from 1 to N in Eq. 3 can be rewritten as the sum of g summations over the computation results in each $\mathcal{G}_i$:

$$\begin{aligned}
&\|\sum_{i=1}^{N}\left(\nabla l_i^+(\theta) - \nabla l^+_{\gamma(\phi_\theta(i))}(\theta)\right)\| \\
=&\|\sum_{i=1}^{g}\sum_{k\in G_i}\left(\nabla l_k^+(\theta) - \nabla l^+_{\gamma(\phi_\theta(k))}(\theta)\right)\| \\
\leq&\sum_{i=1}^{g}\|\sum_{k\in G_i}\left(\nabla l_k^+(\theta) - \nabla l^+_{\gamma(\phi_\theta(k))}(\theta)\right)\| \\
\leq&\sum_{i=1}^{g}|G_i|\max_{k\in G_i}\|\nabla l_k^+(\theta) - \nabla l^+_{\gamma(\phi_\theta(k))}(\theta)\|
\end{aligned} \tag{4}$$

Eq. 4 comes from the triangle equation. More specifically, $\|\nabla l_k^+(\theta) - \nabla l^+_{\gamma(\phi_\theta(k))}(\theta)\|$ denotes the gradient difference between a tuple $t_k^+ \in T^+$ and the tuple $t^+_{\gamma(\phi_\theta(k))}$ that t_k^+ assigns to. Hence, in each group, the sum of gradient difference can be bounded by the group size multiplied by the maximum difference in the group, *i.e.,* $|G_i| \max_{k\in G_i}\|\nabla l_k^+(\theta) - \nabla l^+_{\gamma(\phi_\theta(k))}(\theta)\|$. Thus, Eq. 3 can be bounded by the result of Eq. 4. Next, we focus on how to minimize the bound.

Recap from Example 3, intuitively, the bound (*i.e.,* the right hand in Eq. 4) will be minimized when $\phi_\theta(k)$ is set to assign each t_k^+ to the closest tuple in the coreset C, *w.r.t.* the gradient, as follows:

$$\sum_{i=1}^{g}|G_i|\max_{k\in G_i}\min_{c_j\in C}\|\nabla l_k^+(\theta) - \nabla l^+_{\gamma(j)}(\theta)\| \tag{5}$$

However, given a coreset, it is infeasible to compute Eq. 5 because we have to iterate each tuple in every group, which is equivalent to iterate the entire T^+, but the large T^+ will not be materialized due to the inefficiency. To address this issue, considering Eq. 4 and Eq. 5 and leveraging max-min inequality [5] over Eq. 5, we can get:

$$
\begin{aligned}
&\|\sum_{i=1}^{N}\nabla l_i^+(\theta) - \sum_{j=1}^{|C|} w_j \nabla l_{\gamma(j)}^+(\theta)\| \\
\le &\sum_{i=1}^{g} |G_i| \max_{k\in G_i}\min_{c_j\in C}\|\nabla l_k^+(\theta) - \nabla l_{\gamma(j)}^+(\theta)\| \qquad (6)\\
\le &\sum_{i=1}^{g} |G_i| \min_{c_j\in C}\max_{k\in G_i}\|\nabla l_k^+(\theta) - \nabla l_{\gamma(j)}^+(\theta)\|
\end{aligned}
$$

Given Eq. 6, we can iterate the much smaller coreset C and the gradient approximate error can be bounded using the largest gradient difference between $c_j \in C$ and tuples within each group, *i.e.,* $\max_{k\in G_i}\|\nabla l_k^+(\theta) - \nabla l_{\gamma(j)}^+(\theta)\|$, which can be computed efficiently without joining all tables (see § 5). In this way, all tuples within a group will be assigned to the same tuple in the coreset, *i.e.,* $\forall k \in G_i, \phi_\theta(k) = j$.

So far, the deductions we have discussed only consider the case for a particular θ. Obviously, it is prohibitively expensive to explore every possible θ. Next, we illustrate how to bound the gradient approximation error for the parameter space ϑ.

4.2 Upper Bound for the Parameter Space ϑ and Groups

Fortunately, it has been proved in recent works [4, 17, 32] that for convex ML algorithms, *e.g.,* linear regression, logistic regression, the normed gradient difference between tuples can be efficiently bounded by:

$$\forall i, j, \max_{\theta\in\vartheta}\|\nabla l_i(\theta) - \nabla l_j(\theta)\| \le \max_{\theta\in\vartheta} O(\|\theta\|) \cdot \|\mathbf{x}_i - \mathbf{x}_j\| \qquad (7)$$

where $\|\mathbf{x}_i - \mathbf{x}_j\|$ denotes the Euclidean distance between the feature vectors of two tuples. Since $O(\|\theta\|)$ is a constant, we can conclude that **the gradient approximation error can be bounded independent of the optimization problem in practice, *i.e.,* any particular** θ. Thus, based on the results in Eq. 6, we can get:

$$
\begin{aligned}
&\max_{\theta\in\vartheta}\sum_{i=1}^{g}|G_i|\min_{c_j\in C}\max_{k\in G_i}\|\nabla l_k^+(\theta) - \nabla l_{\gamma(j)}^+(\theta)\| \\
\le &\sum_{i=1}^{g}|G_i|\min_{c_j\in C}\max_{k\in G_i}\max_{\theta\in\vartheta}\|\nabla l_k^+(\theta) - \nabla l_{\gamma(j)}^+(\theta)\| \qquad (8)\\
\le &\underbrace{c}_{const}\cdot\sum_{i=1}^{g}|G_i|\min_{c_j\in C}\max_{k\in G_i}\|\mathbf{x}_k^+ - \mathbf{x}_{\gamma(j)}^+\|
\end{aligned}
$$

Feature similarity. For ease of representation, we use similarity $s_{ji} = 1 - normalized(\max_{k\in G_i}\|\mathbf{x}_k^+ - \mathbf{x}_{\gamma(j)}^+\|)$ to denote the minimum similarity of feature vectors between $c_j \in C$ and tuples in group $\mathcal{G}_i$.

Obviously, the minimization of $\min_{c_j\in C}\max_{k\in G_i}\|\mathbf{x}_k^+ - \mathbf{x}_{\gamma(j)}^+\|$ is equivalent to the maximization of $\max_{c_j\in C} s_{ji}$.

Therefore, the gradient approximation error can be bounded by $\sum_{i=1}^{g}|G_i|\max_{c_j\in C} s_{ji}$.

Note that Eq. 7 holds for tuples with the same label [4, 17]. Hence, we need to select subsets of coresets for tuples with different labels and combine them. For example, given a binary classification task (30% of label 0 and 70% with label 1), to select a coreset with size K, we separately select a coreset of size $30\%K$ for tuples with label 0 and another coreset of size $70\%K$ for tuples with label 1, and then merge them. For regression tasks, we will cluster tuples with similar labels, select subsets of coresets for these clusters and merge them.

Problem (GGEM). The problem shown in Eq. 2 can be converted to the **group-based gradient approximation error minimization (GGEM)** problem as follows:

$$C^* = \underset{C\subseteq T^+}{\arg\max}\sum_{i=1}^{g}|G_i|\max_{c_j\in C} s_{ji}, \text{ s.t. } |C| \le K \qquad (9)$$

At a high level, when $T^+ = T \bowtie T_1 \bowtie \cdots \bowtie T_m$ is too large to compute the coreset, the GGEM problem is to efficiently select an optimal coreset C^* such that the tuples in C^* and their associated weights can well approximate the full gradient in T^+, by grouping T^+ and distributing the computation to multiple tables based on the groups. In § 5, we will show how to compute the partial feature similarity s_{ji} efficiently without the fully materialized T^+. Next, we will prove that even if s_{ji} is known, GGEM problem is NP-hard. We will also discuss its submodular property, based on which a greedy algorithm will be designed in § 5.

Theorem 1. *The* GGEM *problem is NP-hard.*

Proof. We start the proof by considering a special case of GGEM problem where $\forall i \in [1, g], |G_i| = 1$. Thus, the number of groups g equals to the number of tuples N. Therefore, the problem becomes $C^* = \underset{C\subseteq T^+}{\arg\max}\sum_{i=1}^{N}\max_{c_j\in C} s_{ji} = \underset{C\subseteq T^+}{\arg\min}\sum_{i=1}^{N}\min_{c_j\in C}\|\mathbf{x}_i^+ - \mathbf{x}_{\gamma(j)}^+\|, |C| \le K$. Naturally, the K-medoid problem [14] can be reduced to the the special case and thus GGEM is NP-hard. □

Theorem 2. *The* GGEM *problem has the submodular property.*

Proof. We start the proof by considering a special case of GGEM problem where $\forall i \in [1, g], |G_i| = 1$. Thus, the number of groups g equals to the number of tuples N. Therefore, the problem becomes $C^* = \underset{C\subseteq T^+}{\arg\max}\sum_{i=1}^{N}\max_{c_j\in C} s_{ji} = \underset{C\subseteq T^+}{\arg\min}\sum_{i=1}^{N}\min_{c_j\in C}\|\mathbf{x}_i^+ - \mathbf{x}_{\gamma(j)}^+\|, |C| \le K$. Naturally, the K-medoid problem [14] can be reduced to the the special case and thus GGEM is NP-hard. □

Theorem 3. *The* GGEM *problem has the submodular property.*

Proof. We first define the utility function U as $U(C) = \sum_{i=1}^{g}|G_i|\max_{c_j\in C} s_{ji}$. The GGEM problem is to maximize utility U. If U has the submodular property, for $C \subseteq T^+$, we have to prove (1) U is monotonous, *i.e.,* $U(C \cup \{e\}) - U(C) \ge 0$, and (2) U has the diminishing marginal returns property, *i.e.,* $U(C \cup \{e\}) - U(C) \ge U(C^+ \cup \{e\}) - U(C^+)$ for any $C \subseteq C^+ \subseteq T^+$ and $e \in T^+ \setminus C^+$. For simplicity, we use $U(e|C)$ to denote $U(C \cup \{e\}) - U(C)$ in the following parts of the paper. We start the proof by considering s_{ji} is known, which will be computed in § 5. In this situation, each group $\mathcal{G}_i$ will be assigned to a tuple of the coreset that maximizes the utility, *i.e.,* assigned to $\arg\max_j s_{ji}$.

For (1), when e is added into C, if no group will be assigned to e, then $U(C \cup \{e\}) = U(C)$. If one or more groups are assigned to e, clearly $U(C \cup \{e\}) > U(C)$. Hence, U is monotonous.

For (2), we can see that $U(C)$ is the sum of different terms *w.r.t.* different groups, and they are computed independently. Therefore, if there is only a single group and the diminishing marginal returns property satisfies, then U has the property. Suppose that the group is denoted by $\mathcal{G}_*$. Given C, C^+ and $\{e\}$, we prove the diminishing marginal returns for $\mathcal{G}_*$ in all possible three cases of $c_* = \underset{c_j \in C \cup (C^+ \setminus C) \cup \{e\}}{\arg\max} |G_*| s_{j*}$.

[Case 1: $c_* \in C$] In this case, obviously, $U(C \cup \{e\}) - U(C) = U(C^+ \cup \{e\}) - U(C^+) = 0$ because $\mathcal{G}_*$ will not change its assignment when $C^+ \setminus C$ and e are added.

[Case 2: $c_* \in C^+ \setminus C$] Apparently, $U(C^+ \cup \{e\}) - U(C^+) = 0$, which must be smaller than $U(C \cup \{e\}) - U(C)$.

[Case 3: $c_* = e$] There are two cases here.

(1) If $|G_*| \max_{c_j \in C} s_{j*} \geq |G_*| \max_{c_j \in C^+ \setminus C} s_{j*}$, $U(C \cup \{e\}) - U(C) = U(C^+ \cup \{e\}) - U(C^+) > 0$.

(2) If $|G_*| \max_{c_j \in C} s_{j*} < |G_*| \max_{c_j \in C^+ \setminus C} s_{j*}$, $U(C \cup \{e\}) - U(C) > U(C^+ \cup \{e\}) - U(C^+) > 0$. The reason is that when $C^+ \setminus C$ is added to C, $\mathcal{G}_*$ will change its assignment and thus the utility is increased. Afterwards, e is added, and the utility is further increased.

In summary, the GGEM problem has the submodular property. □

Our scope. Note that we focus on the convex problems trained with gradient descent because for such problems, the gradient difference can be bounded by the difference between feature vectors. In this situation, regardless of any convex ML algorithm or parameter θ, RECON can select the coreset without training in advance.

4.3 Connection between Groups of the Single Augmented Table and Multiple Tables

As mentioned in § 3.3, the joined result T^+ is generally large in scale, which makes it inefficient to directly select coreset over T^+. To address this, our key idea is to divide the large-scale T^+ into some disjoint groups (like the Groupby), each of which contains tuples that have the same attribute values over one or more predefined attributes. In this way, the gradient computation over tuples in T^+ can be pushed down as a pre-computation step in the corresponding groups of each single table respectively, and further bounded by aggregating the results from multiple tables efficiently.

Remark. In this paper, we consider the join type that introduces redundancy in the data, including both tuple redundancy and feature redundancy [8, 25] (*e.g.,* PK-FK, one (many)-to-many joins). For example, considering $R = S \bowtie T$, for table S, the tuple ratio is denoted by $\frac{n_R}{n_S}$ and feature ratio is $\frac{d_R}{d_S}$, where n represents the number of tuples and d represents the number of features. We assume that the tuple ratio and the feature ratio are larger than 1.

5 RECON ALGORITHM

Theory 3 in § 4, *i.e.,* the GGEM problem has the submodular property, tells us that we can use a greedy algorithm with the approximate ratio $(1-\frac{1}{e})$ to solve the problem of coreset selection without joining multiple tables. In what follows, we will first overview the algorithm in § 5.1, followed by discussing two important components of the algorithm in § 5.2 and § 5.3 respectively.

Algorithm 2: RECON Algorithm

```
   Input: The train data $T$, $\mathcal{T} = \{T_1, T_2, ..., T_m\}$, coreset size $K$.
   Output: A coreset $C \subseteq T^+$, weight $W = \{w_j\}, |C| = |W| = K$.
1  Pre-compute table-wise partial feature similarity difference
   $D^+ = \{d^h_{ij} | \forall T_h \in \mathcal{T} \cup \{T\}, \forall t^h_i, t^h_j \in T_h, d^h_{ij} = \|\mathbf{x}^h_i - \mathbf{x}^h_j\|\}$.
2  $C = \emptyset$;
3  while $|C| < K$ do
4      Sample a subset $\mathcal{S}$ from $T^+$ using $T$ and $\mathcal{T}$;
5      for each tuple $t_j \in \mathcal{S}$ do
6          $U(C \cup \{t_j\}) = 0$;
7          for each group $\mathcal{G}_i \in \mathcal{G}$ do
8              Compute $s_{ji}$ by aggregating $d^h_{i'j'} \in D^+$ from different
               tables;
9              $U(C \cup \{t_j\})$ += $|G(i)| \max_{c_j \in C \cup \{t_j\}} s_{ji}$ ;
10         $U(t_j|C) = U(C \cup \{t_j\}) - U(C)$;
11     $t^* = \arg\max_{t_j \in \mathcal{S}} U(t_j|C)$ ;
12     Add $t^*$ to $C$ ;
13 for $j$ = 1 to $|C|$ do
14     $w_j = \sum_{i=1}^{g} |G(i)| \mathbb{I}[j = \arg\max_{c_{j'} \in C} s_{j'i}]$
15 return $C, W$;
```

5.1 Algorithm Overview

Algorithm. Algorithm 2 takes the base table T, a set $\mathcal{T}$ of tables to be augmented and the coreset size K (can be specified by the user) as input, greedily adds the tuple that brings about the largest utility improvement into the coreset such that a near-optimal coreset is finally output. Note that T^+ denotes the result of joining all above tables together, which will not be materialized.

It first pre-computes the feature vector difference between every two tuples in each table, so as to bound the gradients (line 1). Next, it iteratively adds one tuple to the coreset at a time (the first loop in lines 3-12). At each iteration, the tuple with the largest utility among T^+ will be selected (line 11). However, it is rather time-consuming to iterate T^+, and thus we can sample a set $\mathcal{S}$ of joined tuples and select one from them (the second for loop in lines 5-10). Afterwards, to compute the utility of each tuple $t_j \in \mathcal{S}$, the third loop (lines 7-9) iterates each group $\mathcal{G}_i$, computes the feature similarity s_{ji} (line 8), reconsiders whether tuples in $\mathcal{G}_i$ should be assigned to t_j, and updates the utility value (line 9). In short, we can derive the coreset C by repeatedly sampling tuples, computing their utilities, and select the best one as a member of the coreset. Finally, we also have to assign each $c_j \in C$ a weight w_j.

Algorithm 2 consists of the following four key components.

[Component ❶: *Partial feature similarity* pre-computation.] As shown in line 1, we first compute the similarity of partial feature vectors in each individual table, in order to bound the gradient based on Eq. 7. This step is easy to implement but rather important. The motivation and the details will be introduced in § 5.2.

[Component ❷: *Joined tuples sampling.*] We use the sampling method proposed in [47] to sample a set $\mathcal{S} \subset T^+$ of tuples as candidates (line 4). Although T^+ will not be materialized, [47] guarantees that these sampled tuples are uniformly sampled from T^+. Thus, there still has an approximate ratio $1 - \frac{1}{e} - \epsilon$ for the greedy algorithm, as discussed in § 3.2, where $|\mathcal{S}| = (N/K) \cdot log(1/\epsilon)$.

[Component ❸: *Feature similarity computation.*] During the above

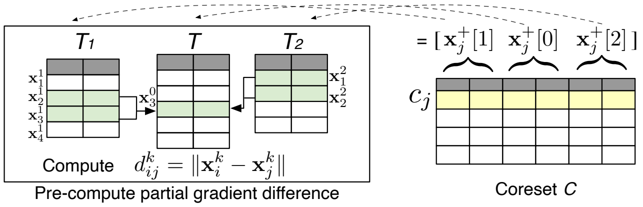

T_1
T
T_2
$\mathbf{x}_1^1$
$\mathbf{x}_2^1$
$\mathbf{x}_3^1$
$\mathbf{x}_4^1$
$\mathbf{x}_3^0$
$\mathbf{x}_1^2$
$\mathbf{x}_2^2$
Compute $d_{ij}^k = \|\mathbf{x}_i^k - \mathbf{x}_j^k\|$
Pre-compute partial gradient difference
= [$\mathbf{x}_j^+[1]$ $\mathbf{x}_j^+[0]$ $\mathbf{x}_j^+[2]$]
c_j
Coreset C

Figure 6: Partial feature similarity pre-computation.

process, computing s_{ji} is challenging because we do not have T^+. To address this , the high level idea is to aggregate the pre-computation results in Component ❶ along with the join keys, which will be introduced in § 5.3.

[Component ❹: *Weight computation.*] As shown in line 14, w_j equals to the sum of the groups that are assigned to c_j timing the group size, because the tuples within a group will be assigned a single tuple in the coreset.

5.2 Partial Feature Similarity Pre-computation

As discussed above, to solve the GGEM problem, it is significant to compute the feature similarity s_{ji}, *i.e.,* the minimum similarity of feature vectors between c_j and tuples in the group $\mathcal{G}_i$, so as to bound the gradient difference. The computation is highly related to the Euclidean distance between two feature vectors, as shown in Eq. 8. Although we can obtain one vector of c_j through sampling, the other one in the group is not available because we do not want to materialize T^+ and iterate it. Fortunately, each feature vector in T^+ can be represented by the concatenation of $(m+1)$ sub-vectors from these m candidate tables as well as the base table, *i.e.,* $\mathbf{x}_i^+ = [\mathbf{x}_i^+[0], \mathbf{x}_i^+[1], ..., \mathbf{x}_i^+[m]]$, where $\mathbf{x}_i^+[0]$ denotes the base table part of feature vector of t_i^+. Intuitively, to capture the feature difference, we can first capture the partial feature similarity inside each table, and then aggregate from multiple tables to compute s_{ji}.

Hence, we push down the computation to each individual table as a pre-computation step, which accelerates the coreset selection much. To be specific, for each table T_k, the feature vector difference of any tuple pair, *i.e.,* $d_{ij}^h = \|\mathbf{x}_i^h - \mathbf{x}_j^h\|$ is computed.

In Figure 6, suppose that the tuples colored green from multiple tables will form a group $\mathcal{G}_2$. In pre-computation, $\|\mathbf{x}_j^+[1] - \mathbf{x}_2^1\|$, $\|\mathbf{x}_j^+[1] - \mathbf{x}_3^1\|$, $\|\mathbf{x}_j^+[0] - \mathbf{x}_3^0\|$, $\|\mathbf{x}_j^+[2] - \mathbf{x}_1^2\|$ and $\|\mathbf{x}_j^+[2] - \mathbf{x}_2^2\|$ have been computed when we want to compute s_{12}, which accelerates the coreset selection much because we do not need to compute the differences over the large scale T^+. Note that we are not going to compute the feature similarity between every two tuples in T^+. Recap that in § 4.2, the similarity s_{ji} to be computed, *i.e.,* $s_{ji} = 1 - normalized(\max_{k \in G_i} \|\mathbf{x}_k^+ - \mathbf{x}_{\gamma(j)}^+\|)$ is to denote the minimum similarity of feature vectors between $c_j \in C$ and tuples in group $\mathcal{G}_i$. Once s_{ji} can be computed, we can use it to bound the gradient approximation error, so in the next section, we will discuss how to aggregate pre-computed partial results to derive s_{ji}.

5.3 Gradient Aggregation for Feature Similarity (s_{ji}) Computation

Next, we describe, for a given $c_j \in C$, how to efficiently compute s_{ji}, which is equivalent to compute the difference, *i.e.,* $\max_{k \in G_i} \|\mathbf{x}_k^+ - \mathbf{x}_{\gamma(j)}^+\|$ for all the groups $\mathcal{G}_i \in \mathcal{G}$ using a dynamic programming (DP) algorithm. We mainly illustrate the algorithm using a concrete example as shown in Figure 7. Before that, we first introduce some necessary notations.

Recap that in § 5.2, each sampled tuple $\mathbf{x}_{\gamma(j)}^+$ can be represented by concatenating m+1 sub-vectors from different tables respectively. Hence, for each subvector $\mathbf{x}_{\gamma(j)}^+[h], h \in [0, m]$, we use $\gamma_h(j)$ to denote the index of $\mathbf{x}_{\gamma(j)}^+[h]$ in table T_h. In this way, for all tuples in each table T_u, $d_{\gamma_h(j),u}^h = \|\mathbf{x}_{\gamma_h(j)}^h - \mathbf{x}_u^h\| = \|\mathbf{x}^+[h] - \mathbf{x}_u^h\|, u \in [1, |T_u|]$. Besides, we model these tables as a tree structure, where the root is the table that we group on. We use J_h to denote the set of children table index of T_h. R_u^h denotes the intermediate result of t_u^h (The u-th tuple in T_h) joining with all the descendants of T_h. Next, we use two examples to clarify these.

Example 4. *[Join tree] Suppose that $\mathcal{A} = \{A_0\}$, we have a tree in Figure 7, where T_0 is the root, and T_1, T_3 are leaves. Thus, $J_0 = \{1, 2\}$, $J_2 = \{3\}$ and $J_1 = J_3 = \emptyset$. Besides, T_0 and T_1 can be joined on attribute A_2, where values with the same color can be joined together. For example, the first two tuples in T_0 can be joined with the first tuple in T_1. Based on that, R_1^2 denotes the result (*i.e., *two tuples) of t_1^2 joining with T_3 (the descendant of T_2). R_1^2 denotes the result (*i.e., *three tuples) of t_1^1 joining with T_1, T_2 and T_3 (the descendants of T_0).*

*[Groups] Suppose that we want to group on attribute A_0 (*i.e., *the group key). Clearly, both T_0 and T^+ will have 3 groups $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$. For example, the first group $\mathcal{G}_1$ in T_0 has two tuples, but after joining, there will be six tuples in the group.*

Our goal. Given the join tree, group key and pre-computation results (§ 5.2), the goal of our DP algorithm is to compute the feature similarity s_{ji} between each group $\mathcal{G}_i$ and the tuple c_j in the coreset, by aggregating the results from multiple tables along with the join keys, without materializing T^+.

Key observation. The feature similarity computation has the optimal substructure. Hence, at a high level, we can group within each individual table on the attributes to be joined, and then use a dynamic programming algorithm to compute the s_{ji} across the join relations. Taking the $\mathcal{G}_1$ as an example, that is, we want to get s_{11} (the similarity/difference between c_1 and $\mathcal{G}_1$). For the two tuples t_1^0 (t_2^0), we have already known $d_{\gamma_0(1),1}^0$($d_{\gamma_0(1),2}^0$). Hence, if we can compute the maximum difference of feature vectors of tuples that can join with t_1^0 (t_2^0) from other tables, which serve as the *optimal substructure*, we can add $d_{\gamma_0(1),1}^0$($d_{\gamma_0(1),2}^0$) to the corresponding maximum difference and output two values. Finally s_{11} can be computed by choosing the largest one from the two values.

Specifically, to capture the join relations of tuples, except the base table, for each table T_h, we group the tuples in T_h on the attribute that serves as the key to join with T_h's parent. We use P_v^h to denote the v-th group of tuples in T_h. For example, P_1^2 includes the first two tuples in T_2 and P_2^3 just includes the last tuple in T_3, as shown in Figure 7. Note that for the base table, the groups are constructed based on A_0 rather than the join key because T_0 is the root.

We use $dp[u, h]$ to denote the maximum difference between tuples in R_u^h and c_j's corresponding sub-vectors *w.r.t.* tuples in R_u^h. We then use $DP[v, h]$ to denote the maximum dp value among the v-th group of T_h, *i.e.,* $DP[v, h] = \max_{t_u^h \in P_v^h} dp[u, h]$. Thus, we have:

$$dp[u, h] = d_{\gamma_h(j),u}^h + \sum_{h' \in J(h)} DP[v', h'] \tag{10}$$

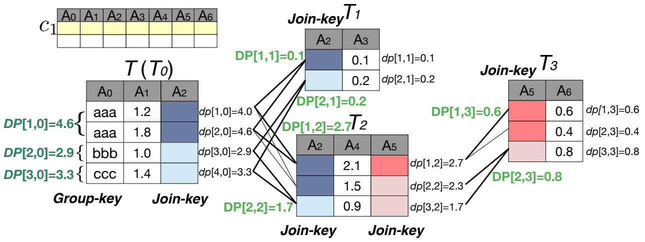


Figure 7: An example of dynamic programming.

where v' denotes the index of group in $T_{h'}$ that can join with t_u^h. Then we can compute $DP[i,0], i \in [1,g]$ following Eq. 10. Obviously, we can directly compute s_{ji} based on $DP[i,0]$.

For ease of discussion, here we just consider equi-join where each tuple can join with at most one group of a table. We will discuss how to support fuzzy join later in this section.

Let us illustrate the algorithm using an example to compute s_{11}.

Example 5. *[The DP algorithm] We run the DP algorithm from bottom to up. Initially, we compute the dp values for all tuples in the leaf nodes,* i.e., *T_1 and T_3 in Figure 7. For example, to compute $dp[1,3]$, since $J(3) = \emptyset$, $dp[1,3] = d^3_{\gamma_3(1),1} = 0.6$ following Eq. 10. Next we consider how to compute $DP[1,3]$. Group P^3_1 consists of t^3_1 and t^3_2 as they have the same value on the join key attribute A_5, so we can compute $DP[1,3] = \max(dp[1,3], dp[2,3]) = 0.6$. Thus, the largest in the group* (i.e., *$dp[1,3]$) will be propagated (marked as the bold line) to its father relation* (i.e., *T_2) for further computation. Then we come to T_2. Similarly, to compute $dp[1,2]$, considering Eq. 10 $d^2_{\gamma_2(1),1}$ is added first. After that, since $J(2) = 3$ and t^2_1 joins with tuples in P^3_1, $DP[1,3]$ will also be added to $dp[1,2]$. Therefore, we obtain $dp[1,2] = d^2_{\gamma_2,1} + DP[1,3] = 2.1 + 0.6 = 2.7$. After that, $DP[1,2]$ is computed as* $\max(dp[1,2], dp[2,2]) = 2.7$*, which is then propagated to the first two tuples in T_0 that join with tuples in P^2_1. Finally, we come to T_0. To compute $dp[1,0]$, $d^0_{\gamma_0(1),1}$ is first added to $dp[1,0]$ as well. Since $J(0) = \{1,2\}$, $DP[1,1]$ and $DP[1,2]$ will be added to $dp[1,0]$. Therefore, we can obtain $dp[1,0] = d^0_{\gamma_0(1),1} + DP[1,1] + DP[1,2] = 1.2 + 0.1 + 2.7 = 4.0$. Finally, $DP[1,0]$ is computed as $DP[1,0] = \max(dp[1,0], dp[2,0]) = 4.6$.*

Complexity Analysis of Algorithm 2. Recap that $|T^+| = N$. For ease of illustration, we use $O(n)$ to denote the average size of each single table, *i.e.,* $O(|T_h|) = O(n)$. Besides, we use D (d) to denote the number of feature dimensions of T^+ (T_h on average) respectively. Next, we analyze the time complexity from two aspects, *i.e.,* the pre-computation and greedy algorithm with 3 loops.

Partial Feature Similarity Pre-computation. In this stage, we need to pre-compute the difference of feature vectors between every two tuples in each table, so the time complexity is $O(n^2 d)$ when the number of tables can be regarded as a constant.

Greedy Coreset Selection. To select the coreset C, the greedy algorithm repeats K times. Each time a set of tuples $\mathcal{S} \subset T^+$ is sampled. To get i.i.d. uniform samples, we use the exact weight algorithm from Zhao et al. [47], which has no accept-reject step in the sampling phase. After a pre-computation in $O(n)$, we can get a sample from T^+ in $O(1)$ [44, 47]. For each element $t_j \in \mathcal{S}$, we need to compute $U(t_j|C)$, which involves the computation of s_{ji} for every group $\mathcal{G}_i \in \mathcal{G}$ using the DP algorithm. The complexity of DP is linear to the total sizes of relations, *i.e.,* $O(|T_0| + |T_1| + \cdots + |T_m|) = O(n)$. For ease of representation, we use S to denote $|\mathcal{S}|$. Therefore, The time complexity of this part is $O(K \cdot |\mathcal{S}| \cdot n) = O(nKS)$.

Total Time Complexity. In summary, the total time complexity of our approach is $O(n^2 d + nKS)$.

To show the superiority of our method, we also illustrate the time complexity of the SOTA single-table coreset selection algorithm on T^+, if the join result is materialized. First, they compute feature vector differences between every two tuples in T^+, so as to bound the gradient, leading to a time complexity of $O(N^2 D)$. Afterwards, to compute the utility of each tuple , their methods have to iterate every tuple in T^+, leading to a time complexity of $O(K \cdot S \cdot N) = O(NKS)$. In total, the time complexity is $O(N^2 D + NKS)$.

For feature-enrich ML, $N \gg n$ always holds when various types of joins exist. Thus, our method can much improve the efficiency.

Discussion. For fuzzy join, each tuple t_u^h may join with multiple groups $P^{h'}_{v'}$ for $h' \in J(h)$ of a table. We can extend RECON to handle this by changing the $DP[v', h']$ in Eq. 10 into $dp[u,h] = d^h_{\gamma_h(j),u} + \sum_{h' \in J(h)} \max_{v' \in V} DP[v', h']$, where V represents all the groups in $T_{h'}$ that can join with t_u^h. For the grouping key, if the attributes in $\mathcal{A}$ specified by the user are distributed in multiple tables, we can run our algorithm by randomly selecting a table from these tables as the root and aggregating the results using the DP algorithm.

Note that the upper bound of the gradient difference derived in § 4 only holds for points with similar labels. Thus, theoretically we need to select subsets separately. The above analysis assumes that all tuples correspond to same labels. However, in practice, data has different labels. Therefore, data with a certain label generally represents only a small fraction of the total data. The total amount of calculation is much less than the above complexity.

Convergence Rate Analysis of Algorithm 2. In the field of ML, convergence rate reflects how fast the machine learning algorithm can find the optimal parameters. The higher the convergence rate, the fewer iterations the model needs to converge. Specifically, we can compute the convergence rate by comparing the parameter θ computed in the k-th and the $(k+1)$-th iteration to the optimal one. Recap that l is a strongly convex function. Thus, $\forall \theta, \theta'$ we have

$$l(\theta) \geq l(\theta') + \nabla l(\theta')(\theta - \theta') + \frac{\mu}{2}\|\theta - \theta'\|^2 \tag{11}$$

where μ is a constant. Recap that in § 4, the gradient approximate error can be bounded, *i.e., $max_{\theta \in \vartheta}$*$\| \sum_{i=1}^{N} \nabla l_i^+(\theta) - \sum_{j=1}^{|C|} w_j \nabla l^+_{\gamma(j)}(\theta)\| \leq \epsilon_1$. In addition, we also have $\|\theta_k - \theta_*\| \leq \epsilon_2$. Recap that we use incremental gradient method in Algorithm 2, we denote the stepsize as $\alpha_k = \frac{\alpha_0}{k^\tau}$ (τ is a constant) for the k-th epoch.

Each time using gradient descent, we have $\|\theta_{k+1} - \theta_*\|^2 = \|\theta_k - \alpha_k \sum_{j \in C} w_j \nabla l^+_{\gamma(j)}(\theta^k_{j-1}) - \theta_*\|^2$. Then, following Eq. 11, we have

$$\begin{aligned} \|\theta_{k+1} - \theta_*\|^2 \leq \|\theta_k - \theta_*\|^2 - 2\alpha_k \sum_{j \in C} (l_j^+(\theta_k) - l_j^+(\theta_*)) \\ +2\alpha_k \sum_{j \in C} (l_j^+(\theta^k_{j-1}) - l_j^+(\theta_k)) + \alpha_k^2 \sum_{j \in C} \|w_j \nabla l_j(\theta^k_{j-1})\|^2 \end{aligned} \tag{12}$$

For the item $-2\alpha_k \sum_{j \in C}(l_j^+(\theta_k) - l_j^+(\theta_*))$, after applying Eq. 11, we can get $-2\alpha_k \sum_{j \in C}(l_j^+(\theta_k) - l_j^+(\theta_*)) \leq -2\alpha_k(\sum_{j \in C} w_j \nabla l_j^+(\theta_*)(\theta_k - \theta_*) + \frac{\mu}{2}\|\theta_k - \theta_*\|^2)$. Then, using Cauchy-Schwarz inequlity [43], we can find that:

$$\begin{aligned} & -2\alpha_k \sum_{j \in C}(l_j^+(\theta_k) - l_j^+(\theta_*)) \\ \leq & -\mu\alpha_k \|\theta_k - \theta_*\|^2 + 2\alpha_k \|\sum_{j \in C} w_j \nabla l_j^+(\theta_*)\| \|(\theta_k - \theta_*)\| \\ \leq & -\mu\alpha_k \|\theta_k - \theta_*\|^2 + \frac{2\alpha_k |C| \epsilon_1 \epsilon_2}{\mu} \end{aligned} \tag{13}$$

For item $2\alpha_k \sum_{j \in C}(l_j^+(\theta_{j-1}^k) - l_j^+(\theta_k))$ and $\alpha_k^2 \sum_{j \in C} \|w_j \nabla l_j(\theta_{j-1}^k)\|^2$, since l is strongly convex and assume that $max_{j \in C}\|\nabla l_j(\theta)\| \leq \epsilon_3$, we have

$$\begin{aligned} & 2\alpha_k \sum_{j \in C}(l_j^+(\theta_{j-1}^k) - l_j^+(\theta_k)) + \alpha_k^2 \sum_{j \in C} \|w_j \nabla l_j(\theta_{j-1}^k)\|^2 \\ \leq & 2\alpha_k \sum_{j \in C} \|w_j \nabla l_j^+(\theta_k)\| \alpha_k \sum_{i=1}^{j-1} \|w_i \nabla l_i^+(\theta_{i-1}^k) + \alpha_k^2 \sum_{j \in C} \|w_j \nabla l_j(\theta_{j-1}^k)\|^2 \\ \leq & 2\alpha_k^2 (|C|^2 - |C|) w_{max}^2 \epsilon_3^2 + \alpha_k^2 |C| w_{max}^2 \epsilon_3^2 \end{aligned} \tag{14}$$

Thus, applying Eq. 13 and Eq. 14 to Eq. 12, we can get

$$\|\theta_{k+1} - \theta_*\|^2 \leq (1 - \mu\alpha_k)\|\theta_k - \theta_*\|^2 + \frac{2\alpha_k |C| \epsilon_1 \epsilon_2}{\mu} + \alpha_k^2 |C|^2 w_{max}^2 \epsilon_3^2 \tag{15}$$

Finally, by applying Lemma 4 in [10], the convergence rate of Algorithm 2 is at the same rate of $O(\frac{1}{\sqrt{k}})$ as the convergence rate for incremental gradient descent on the full data T^+ [35]. Hence, theoretically, RECON needs the same number of epochs to converge as training on the full data. In this situation, since the coreset is much smaller than the full data, the efficiency is much improved.

6 EXPERIMENTS

The key questions we seek to answer are: (1) How does RECON perform to select a well-performed coreset with an appropriate size, as an end-to-end solution (§ 6.2)? (2) What about the effectiveness and efficiency of RECON, compared with baselines (§ 6.3 - § 6.6)? (3) Is RECON sensitive to ML models (§ 6.7)?

6.1 Experimental Settings

Dataset. We used 5 widely-used real-world datasets that covered various data characteristics, *e.g.,* the dataset size varying from the magnitude of 10^4 to 10^7. Table 1 shows the statistics of the datasets.

(1) Brazil [1] is a dataset with a multi-classification task to predict *"the review score of an order given by the customer"* with four tables.

(2) IMDB [27] is a dataset that *"predicts the score of movies"* with 7 tables. Obviously, we can regard it as a regression task to predict the score. To show more thorough experiments, similar to [8], we also regard it as a classification task by dividing the rating scores in to 5 equal intervals (*i.e.,* grades) and predict the grade.

(3) IMDB-Large is similar to IMDB, except that IMDB-Large uses all tuples in Cast_info, producing 21,303,410 tuples for T^+.

Table 1: Statistics of datasets.

Dataset	# Tables	# Rows (T^+)	# features (T^+)	Task
Brazil	4	98,463	9	Class.
IMDB	7	674,466	41	Class./Reg.
IMDB-Large	7	21,303,410	41	Class./Reg.
Stack	3	6,347,553	178	Reg.
Taxi	5	2,792,376	30	Reg.

(4) Stack [30] contains questions and answers from the StackExchange, which *"predicts the reputation of users"* as a regression task.

(5) Taxi [9] is to *"predict the number of vehicle collisions in New York City for each day"*. In particular, we have *fuzzy join* on this dataset, *e.g.,* the Weather table can join with the base table on the attribute *w.r.t.* time, but the weather data is represented by the granularity of minutes, hours, or days.

Following [26], for every dataset, the base table is randomly shuffled and divided into 50%/25%/25% proportions as train/validation/test set. All other tables will be used for feature augmentation via joins while training, validating and testing.

The group key $\mathcal{A}$ of each dataset is specified by the user. Although the user can specify any set of attributes as the group key, by default, we use the (primary) key of the base table, if available. We used {review_id}, {movie_id}, {movie_id}, {user_id} and {datetime} as group keys for Brazil, IMDB, IMDB-Large, Stack and Taxi in their base tables respectively.

Baselines. We compared with several baselines.

(1) Base uses the base table T as train data to train ML models (see *e.g.,* Figure 3(a)–❶).

(2) Full uses the fully augmented table T^+ as the train data to train ML models (see *e.g.,* Figure 3(a)–❸).

(3) Sample-Join [47] uniformly samples tuples from T^+ as train data without materializing the join result.

(4) Join-Coreset [32, 34] selects the coreset over fully materialized join result T^+ and uses the coreset as train data (see *e.g.,* Figure 3(a)–❺). We use the popular single-table coreset selection algorithm [32] that follows the paradigm in Figure 5(b).

(5) Coreset-Join first selects a coreset from the base table T, then joins with tables in $\mathcal{T}$ and finally trains on the join result.

(6) FML [8, 25, 41] (factorized ML) focuses on accelerating batch gradient descent algorithm by decomposing the ML computations through joins. Among these methods, [8] is a general one for different ML algorithms, so we compare with it in § 6.6. In other sections, we focus on the stochastic gradient descent algorithm that is widely used in practice due to its high efficiency.

Hyper-parameter Setting. For the classification task and regression task, we train logistic regression and linear regression models by default respectively, where L2-regularization (regularization coefficient=10^{-5}) and stochastic gradient descent (SGD) are applied. The influence of using different ML models will be evaluated in § 6.3. For training, we fix the number of training epochs to 20. We use *k-inverse decay scheduling, i.e.,* $\alpha_k = \alpha_0/(1 + bk)$, where α_0 and b are tuned as hyperparameters. The sample size S is set to 500. The influence of different sample sizes will be evaluated in § 6.3.

Evaluation Metrics. We evaluate the efficiency in an end-to-end way, including both the consuming time of coreset selection and model training. For effectiveness, we use different evaluation metrics for different tasks. For classification tasks, following previous works [9, 47], we use *model prediction accuracy* as evaluation metric.

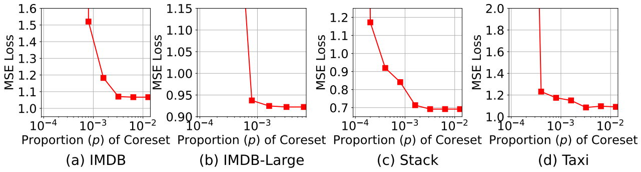


Figure 8: End-to-end coreset selection for regression.

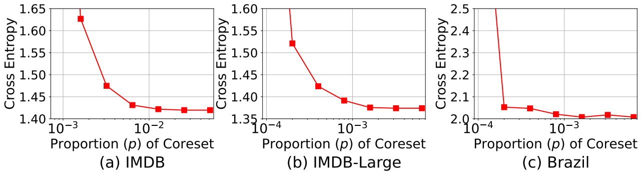


Figure 9: End-to-end coreset selection for classification.

For regression tasks, following [25], we use *root mean squared error* (*RMSE* = $\sqrt{\frac{\sum_{t=1}^{N}(\hat{y}_t - y_t)^2}{N}}$) as evaluation metric.

Remark. Note that joins may change the original distribution of the base table T, *i.e.,* one tuple in T may correspond to multiple tuples in T^+, and thus the effectiveness can be evaluated both on T or T^+. Evaluation results on both these two distributions are reported in § 6.3 and § 6.4.

6.2 End-to-end Coreset Selection

Recap that in § 5, the proposed algorithm takes as input a user-specified K as the size of the coreset. To realize an end-to-end solution, one may consider how to choose an appropriate size of the coreset. In this part, we propose a simple yet effective approach to achieve this. For ease of explanation, we introduce $p = \frac{K}{N}$ to denote the proportion of coreset compared to the full data in size. N, *i.e.,* $|T^+|$ can be computed efficiently using [47] before it is materialized.

Approach. We start from a coreset in a small size, train on it and evaluate on the validation set, enlarge the coreset and iteratively train until the performance cannot improve much. Specifically, we start from $K = 10^{-4}N$, *i.e.,* $p = 10^{-4}$ and train an initial model. Then, we iteratively enlarge the coreset by 2 times and train. To evaluate each coreset, we apply the model on the validation set and compute the validation loss. If the loss decreases and remains stable within several successive iterations, we stop enlarging the coreset.

Validation loss. Figure 8 -9 show the validation loss (*i.e.,* MSE loss for regression task and cross entropy loss for classification task on the y-axis) by varying the coreset size (the x-axis). At a high level, with the number of tuples of a coreset increasing, the validation loss decreases rapidly first and then remains stable. As Figure 8(c) shows, on Stack dataset, when $K = 0.0032 \times N = 10156$, the loss is 0.69 and then it does not decrease much. We set that within three successive iterations, if the loss varies no more than 1%, we can stop. So finally, we can return the coreset with a size of 10156.

Efficiency. One may consider whether the end-to-end coreset selection including iterative training is time-consuming. The answer is No. More concretely, with the coreset size increasing, it obviously spends more time because both the iterative training and coreset selection consume time, but it is still efficient. For example, it takes 13.3 mins, 4.6 mins, and 1.1 mins on the task of IMDB-Large and Taxi for regression, and Brazil for classification respectively to perform the end-to-end coreset selection. However, if train on T^+ (*i.e.,*Full), it requires 782 mins, 72 mins, and 18 mins respectively. The reasons are (1) The size of coreset is small and thus efficient to train. (2) The number of iterations is not large, and we can fine-tune the model in last iteration without training from scratch. (3) Coreset selection algorithm is efficient and can be done incrementally.

Summary. This end-to-end coreset selection approach provides a way to select an appropriate coreset size. Although several iterations are needed, it is also efficient because the coreset size is small and the coreset selection process is fast.

6.3 Comparison with Baselines

Using the coreset size of each dataset selected in § 6.2, we compare efficiency and effectiveness with baselines. To achieve a fair comparison, for the baselines that sample a subset of tuples to train, we sample the same number of tuples as the coreset size. For Coreset-Join, we set the proportion of coreset over the base table with the same p as RECON.

Efficiency. We show the total time including both coreset selection and model training for different baselines and RECON in Figure 10, given the coreset with the best size. We can see that in general, RECON achieves efficiency improvement nearly two orders of magnitudes compared with Full and Join-Coreset. For example, on dataset IMDB-Large for regression, RECON takes 13.3 minutes, which is nearly 2 orders of magnitudes more efficient than Full (782 mins) and Join-Coreset (612 mins). For classification, on dataset IMDB-Large, RECON takes 110 min, which is also almost 2 orders of magnitudes more efficient than Full (2.8 days) and Join-Coreset (2.2 days). In addition, on dataset Stack and Taxi for regression, RECON takes 11.8 mins and 4.6 mins respectively, which is still an order of magnitude faster than Full (4 hours, 72 mins) and more efficient than Join-Coreset (0.5 hours, 9 mins) over 2 times. The reason is that Full has to train over a large amount of training tuples, *i.e.,* T^+. Although Join-Coreset trains over a coreset with the same size of RECON, it selects the coreset based on T^+, which is rather inefficient. RECON outperforms them because it computes the coreset directly from these tables to be joined. Besides, RECON takes a slightly longer time than other baselines, *e.g.,* Sample-Join (40 mins), Coreset-Join (35 mins) and Base (11 mins) on IMDB-Large for classification, because Base does not need to augment features and select the coreset, Sample-Join just uniformly samples without considering the gradients and Coreset-Join computes the coreset over the much smaller base table T. But they cannot achieve high accuracy as discussed next.

Effectiveness. For regression tasks, on dataset IMDB-Large, we find in § 6.2 that $p = 0.0016$ is the best choice. In this situation, we can observe in Figure 11(b) that RECON has an RMSE of 0.998, which outperforms Sample-Join (1.546). The reason is that Sample-Join just samples for training without considering the gradient approximation. RECON outperforms Base because more useful features are augmented. Besides, RECON also outperforms Coreset-Join (1.603) because the selected coreset of Coreset-Join does not consider features to be augmented. Furthermore, RECON almost has the same RMSE as Join-Coreset (0.988) and Full (0.985) because RECON can well approximate the full gradient accurately with theoretical guarantees. That is, training on the coreset (only 0.0016 proportion of the full data) can achieve the almost same performance as training over the full data. For classification tasks, we have similar observations. For example, on dataset IMDB-Large, we can observe from

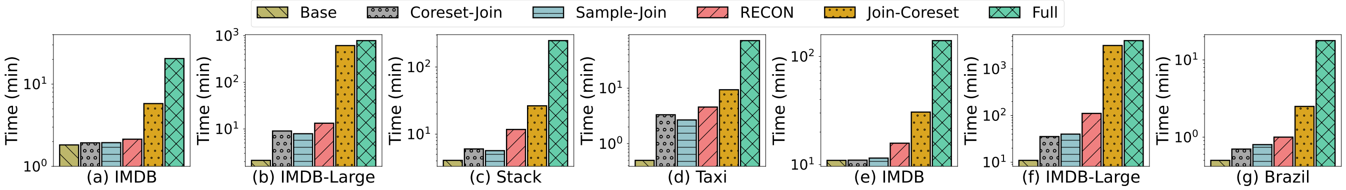


Figure 10: Efficiency. (a,b,c,d): Regression tasks; (e,f,g): Classification tasks.

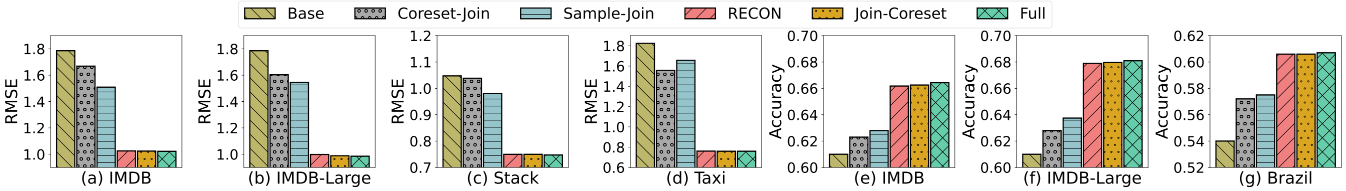


Figure 11: Effectiveness. (a,b,c,d): Regression tasks; (e,f,g): Classification tasks.

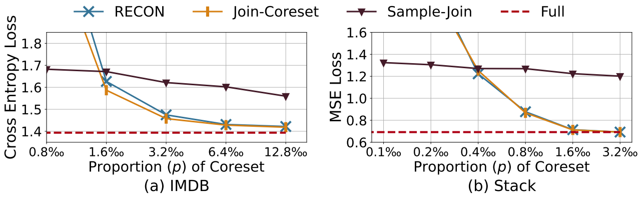


Figure 12: Training loss comparison for IMDB and Stack.

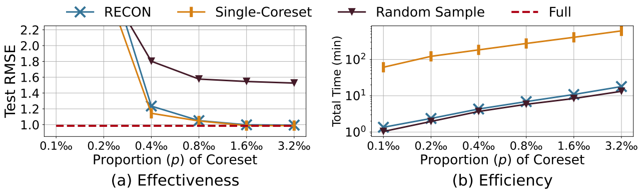
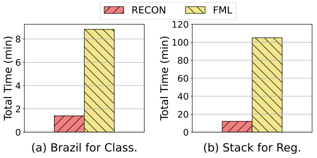


Figure 13: Effectiveness and efficiency by varying p.



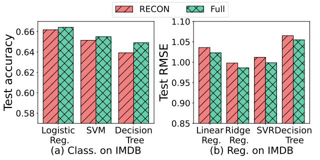


Figure 14: Compare with FML. **Figure 15: Different models.**

Table 2: Convergence results for regression

	# iterations to converge		Test RMSE after convergence	
Dataset	RECON	Full	RECON	Full
IMDB	37,000	6,200,700	1.025	1.023
IMDB-Large	330,000	201,300,000	0.998	0.985
Stack	174,700	62,912,000	0.749	0.747
Taxi	72,000	20,149,000	0.761	0.760

Figure 11(f) that RECON has an accuracy of 0.679, which is higher than Base (0.607), Coreset-Join (0.628), Sample-Join (0.637), and also close to Join-Coreset (0.679) and Full (0.681).

Loss. We show the training loss for IMDB of classification task and Stack of regression task in Figure 12. We can see that RECON converges to almost the same loss as Full, which demonstrates that RECON can accurately estimate the gradient with theoretical guarantees, and thus achieve the same performance as the full data.

Varying the coreset size. We also evaluate the effectiveness and efficiency by varying p in Figure 13 on IMDB-Large of regression task. We only compare RECON with Sample-Join and Join-Coreset, because only they can generate training data of different given sizes. The result of Full is also plotted as a comparison.

Figure 13(a) shows that RMSE of RECON decreases rapidly first, and then remains stable when approaching the best coreset size. RECON outperforms Sample-Join a lot and almost has the same performance as Join-Coreset on all p because RECON can approximate the full gradient well. For efficiency, in Figure 13(b), RECON is more than one order of magnitude faster than Join-Coreset because the coreset selection of RECON has a lower time complexity than Join-Coreset. RECON is only a little slower than Sample-Join because coreset selection of RECON often takes up a small proportion of time compared with iterative training.

Summary. RECON achieves much acceleration (because it computes the coreset without fully materializing the augmented table) for feature-rich ML without sacrificing much effectiveness (because it has the theoretical guarantee on the gradient computation). In addition, we also test on the DNN model and find similar observations, although there is no theoretical guarantees about the gradient. Due to the space limitation, we do not report the results here.

6.4 Comparison with Baselines on Base Table

Different joins may change the underlying distributions. In fact, the accuracy of different baselines can be compared both on the augmented table T^+ or the base table T. Hence, we also add the results with respect to the base table T. To this end, we compare the accuracy between the single table (Base) and multiple tables (all other baselines) as follows. As a tuple $t \in T$ might be joined with multiple tuples in other tables, the tuple will correspond to multiple ones in T^+. Afterwards, for testing the accuracy, we average all the predictions of tuples in T^+ corresponding to t for regression tasks. For classification tasks, we use the majority voting to aggregate the results. Then we compare different methods using the same metrics as Section 6.1. The results are shown in Figure 16.

We can see that after aggregation, RECON still outperforms other baselines. For regression tasks, test RMSE of IMDB-Large for Base is 1.786, while the error of RECON after aggregation is 1.099. For classification tasks, test accuracy of IMDB-Large for Base is 0.61, while the test accuracy of RECON after majority voting is 0.66. This further verifies that feature augmentation can improve the performance.

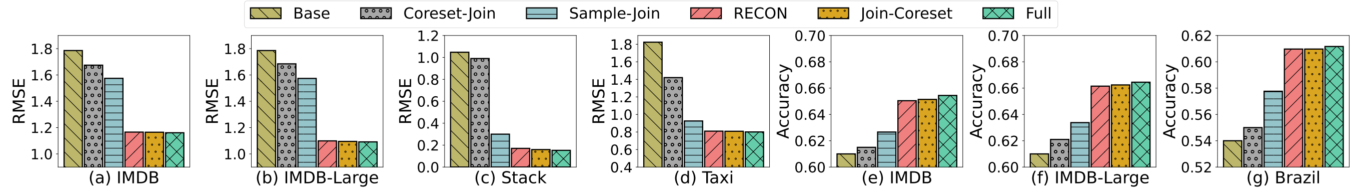


Figure 16: Effectiveness Base by aggregation on T^+. (a,b,c,d): Regression tasks; (e,f,g): Classification tasks.

Table 3: Convergence results for classification

	# iterations to converge		Test accuracy after convergence	
Dataset	RECON	Full	RECON	Full
IMDB	20,500	6,401,000	0.662	0.664
IMDB-Large	340,000	201,220,000	0.679	0.681
Brazil	780	874,800	0.606	0.607

6.5 Convergence Evaluation

In § 5.3, we have theoretically proved the convergence rate of RECON. To empirically test the convergence of different methods, we compute test performance with the increase of SGD iterations. The concrete numbers of iterations to converge and the converged test performance are reported in Table 2 and Table 3. For all the datasets, training on coresets (RECON) converges much faster than Full. From the results in Table 2 and Table 3, we can observe that the speedup of RECON is generally more than two orders of magnitudes. For example, on Stack for regression, training on the coreset (RECON, only 0.0032 proportion of the full data) converges in 174,700 iterations, which is 360 times faster than Full (62,912,000 iterations). In addition, the speedup does not affect the test performance after convergence much, *e.g.,* on IMDB for classification, the test accuracy after convergence of RECON (0.662) is similar to Full (0.664). RECON converges fast with high accuracy, because the coreset selected by RECON is much smaller than the full data, while still approximating the full gradient with theoretical bound.

6.6 Comparison with FML

FML only supports batch gradient training, so we also use batch gradient descent to train for a fair comparison. Since FML aims to accelerate the training process over the full data, it has the same performance as Full, so we only compare the efficiency with FML. We compare the total time including both coreset selection and model training between RECON and FML on Brazil and Stack. Figures 14(a)-(b) report the result, which shows that RECON outperforms FML on both datasets. This is because although FML improves the efficiency by reducing the linear algebra computations, it still needs training over the full data, while our method trains over the judiciously selected coreset with a small size.

6.7 Evaluation of Different ML Models

In § 4.2, we show that the coreset selected by RECON can provide theoretical guarantees for convex models. Here, we test different convex models for an ablation study. For classification, we test logistic regression and support vector machine (SVM) [11]. For regression, we test linear regression, ridge regression and support vector regression (SVR) [12]. Besides, we also test decision tree [37] to evaluate non-convex models. To compare the performance, we evaluate different models on the selected coreset and the full data.

The results are shown in Figure 15(a) and Figure 15(b), where the x-axis denotes different models, and the y-axis denotes the test performance. To be specific, for classification on IMDB, RECON achieves an accuracy of 0.662, 0.651 and 0.640 for logistic regression, SVM and decision tree, which is almost the same as Full (0.664, 0.655, 0.649). For regression on IMDB, RECON achieves an RMSE of 1.036, 1.025, 1.012 and 1.065 for linear regression, ridge regression, SVR and decision tree respectively, which has no significant difference compared with Full (1.027, 1.023, 1.005, 1.054). We can observe that although different models may have different ultimate performance, the performance differences between RECON and Full of different models are not significant. Therefore, these results suggest that RECON can efficiently select coresets that can accurately approximate full gradient regardless of user-specified ML model.

Summary. RECON is not sensitive to underlying ML models because the gradient approximation error can be bounded by the feature similarity in advance rather than the gradients computed during the model training.

6.8 DNN Evaluation

Deep neural networks (DNNs) are currently the most widely-used machine learning algorithm. Despite RECON does not hold theoretical guarantee for DNNs, we have also added new experiments to test the performance of RECON for DNNs.

At a high level, we use RECON to select coresets on the same datasets. We then train DNNs using SGD and test the performance. The coreset sizes and evaluation metrics are the same as that of Section 6.1- 6.3. We first introduce the network architectures of DNNs, and then report the experimental results with analysis.

DNN architecture for classification tasks: We use fully connected networks of 2 hidden layers with 200 nodes for each layer and a softmax output layer. ReLU is used as the activation function. L2 regularization with a coefficient=10^{-4} is used.

DNN architecture for regression tasks: We use fully connected networks with 2 hidden layers, where each hidden layer has 200 nodes and the output layer has one node. ReLU is used as the activation function. L2 regularization with a coefficient = 10^{-4} is used.

The experimental results are shown in Figure 17 and Figure 18 for the effectiveness and efficiency respectively.

Effectiveness. On dataset IMDB-Large (a regression task), we still use $p = 0.0016$ for the coreset size. We can observe in Figure 17(b) that RECON has an RMSE of 0.975, which outperforms Sample-Join (1.128). Also, RECON almost has the same RMSE as Full (training on the fully augmented table T^+, 0.959) and Join-Coreset (training on the coreset selected by [32] from T^+, 0.972).

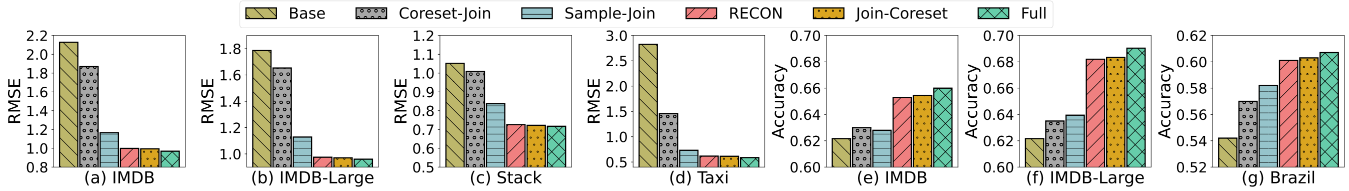


Figure 17: Effectiveness on DNNs. (a,b,c,d): Regression tasks; (e,f,g): Classification tasks.

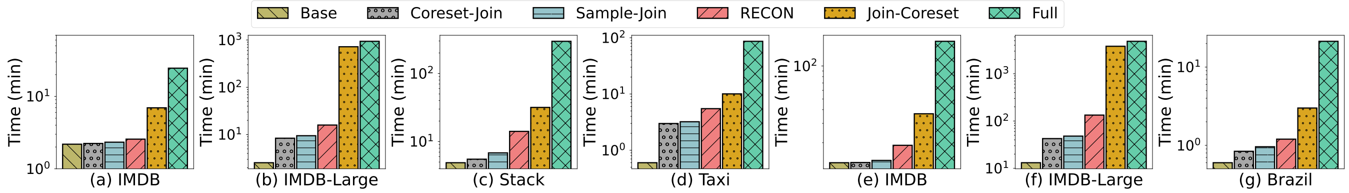


Figure 18: Efficiency on DNNs. (a,b,c,d): Regression tasks; (e,f,g): Classification tasks.

For classification tasks, we have similar observations. For example, on dataset IMDB-Large, we can observe from Figure 17(f) that RECON has an accuracy of 0.682, which is higher than Sample-Join (0.639), and also close to Join-Coreset (0.683) and Full (0.690).

The above results tell us that training on the coreset (only 0.0016 proportion of the full data) selected by RECON can achieve almost the same performance as training over the full augmented data, which empirically shows the superiority of RECON even for DNNs. This is because despite without theoretical guarantee for DNNs, the tuples in the selected coreset can still well represent the full augmented data from the feature similarity perspective, and thus leads to good performance.

Efficiency. We show the total time including both coreset selection and DNN model training for different baselines and RECON in Figure 18. We can see that in general, all methods take more time because DNNs have more parameters to train. However, RECON still achieves efficiency improvement nearly two orders of magnitudes compared with Full and Join-Coreset. For example, on dataset IMDB-Large for regression, as shown in Figure 18(b), RECON takes 16 minutes, which is nearly 2 orders of magnitudes more efficient than Full (900 mins) and Join-Coreset (722 mins). The reason is that Full has to train over a large amount of training tuples, *i.e.,* T^+. Although Join-Coreset trains over a coreset with the same size of RECON, it selects the coreset based on T^+, which is rather inefficient. RECON outperforms them because it computes the coreset directly from these tables to be joined. Besides, RECON takes a slightly longer time than other baselines, *e.g.,* Sample-Join, Coreset-Join and Base, but they cannot achieve high accuracy because they do not augment features or the selected tuples cannot well represent the fully augmented table.

Summary. RECON can still accelerate feature-rich machine learning without sacrificing the effectiveness for DNNs, although there is a lack of theoretical guarantees about the gradient approximation.

7 CONCLUSION

We propose RECON for selecting a coreset of train tuples from an augmented table without materializing it through joins. RECON solves the problem that coreset selection over a big augmented table is time-consuming. The coreset can speed up iterative gradient methods for training ML models (*i.e.,* data-efficient). The augmented features can improve the accuracy of trained ML models (*i.e.,* feature-rich). Extensive experiments verified RECON can much improve the efficiency of coreset selection without sacrificing the performance.

REFERENCES

[1] https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/, 2022. Accessed: 2022-04-28.

[2] A. Adadi. A survey on data-efficient algorithms in big data era. *J. Big Data*, 8(1):1–54, 2021.

[3] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha. Efficient machine learning for big data: A review. *Big Data Res.*, 2(3):87–93, 2015.

[4] Z. Allen-Zhu, Y. Yuan, and K. Sridharan. Exploiting the structure: Stochastic gradient methods using raw clusters. *Advances in Neural Information Processing Systems*, 29, 2016.

[5] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[6] V. Braverman, D. Feldman, and H. Lang. New frameworks for offline and streaming coreset constructions. *CoRR*, abs/1612.00889, 2016.

[7] T. Campbell and T. Broderick. Bayesian coreset construction via greedy iterative geodesic ascent. In *ICML 2018*, volume 80, pages 697–705. PMLR, 2018.

[8] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *arXiv preprint arXiv:1612.07448*, 2016.

[9] N. Chepurko, R. Marcus, E. Zgraggen, R. C. Fernandez, T. Kraska, and D. R. Karger. ARDA: automatic relational data augmentation for machine learning. *Proc. VLDB Endow.*, 13(9):1373–1387, 2020.

[10] K. L. Chung. On a stochastic approximation method. *The Annals of Mathematical Statistics*, pages 463–483, 1954.

[11] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[12] H. Drucker, C. J. Burges, L. Kaufman, A. Smola, and V. Vapnik. Support vector regression machines. *Advances in neural information processing systems*, 9, 1996.

[13] D. Feldman. Introduction to core-sets: an updated survey. *CoRR*, abs/2011.09384, 2020.

[14] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[15] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

[16] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or MAD skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[17] T. Hofmann, A. Lucchi, S. Lacoste-Julien, and B. McWilliams. Variance reduced stochastic gradient descent with neighbors. *Advances in Neural Information Processing Systems*, 28, 2015.

[18] J. Huang, R. Huang, W. Liu, N. M. Freris, and H. Ding. A novel sequential coreset method for gradient descent algorithms. In *ICML 2021*, volume 139, pages 4412–4422. PMLR, 2021.

[19] R. K. Iyer and J. A. Bilmes. Submodular optimization with submodular cover and submodular knapsack constraints. In *NeurIPS 2013*, pages 2436–2444, 2013.

[20] D. Justo, S. Yi, L. Stadler, N. Polikarpova, and A. Kumar. Towards a polyglot framework for factorized ml. *Proc. VLDB Endow.*, 14(12):2918–2931, 2021.

[21] K. Killamsetty, D. Sivasubramanian, G. Ramakrishnan, A. De, and R. K. Iyer. GRAD-MATCH: gradient matching based data subset selection for efficient deep model training. In *ICML 2021*, volume 139, pages 5464–5474.

[22] K. Killamsetty, D. Sivasubramanian, G. Ramakrishnan, and R. K. Iyer. GLISTER: generalization based data subset selection for efficient and robust learning. In *AAAI 2021*,, pages 8110–8118. AAAI Press, 2021.

[23] K. Kirchhoff and J. A. Bilmes. Submodularity for data selection in machine translation. In *EMNLP 2014*, pages 131–141. ACL, 2014.

[24] A. Kumar, M. Jalal, B. Yan, J. F. Naughton, and J. M. Patel. Demonstration of santoku: Optimizing machine learning over normalized data. *Proc. VLDB Endow.*, 8(12):1864–1867, 2015.

[25] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD 2015*, pages 1969–1984, 2015.

[26] A. Kumar, J. F. Naughton, J. M. Patel, and X. Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD 2016*, pages 19–34. ACM, 2016.

[27] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[28] S. Li, L. Chen, and A. Kumar. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In *SIGMOD 2019*, page 1571–1588, 2019.

[29] J. Liu, C. Chai, L. Luo, J. Feng, L. Yin, and N. Tang. Feature augmentation with reinforcement learning. In *ICDE*, 2022.

[30] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *SIGMOD 2021*, pages 1275–1288, 2021.

[31] B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrák, and A. Krause. Lazier than lazy greedy. In *AAAI*, volume 29, 2015.

[32] B. Mirzasoleiman, J. A. Bilmes, and J. Leskovec. Coresets for data-efficient training of machine learning models. In *ICML 2020*, volume 119, pages 6950–6960, 2020.

[33] B. Mirzasoleiman, K. Cao, and J. Leskovec. Coresets for robust training of deep neural networks against noisy labels. In *NeurIPS 2020*, 2020.

[34] A. Munteanu and C. Schwiegelshohn. Coresets-methods and history: A theoreticians design pattern for approximation and streaming algorithms. *KI-Künstliche Intelligenz*, 32(1):37–53, 2018.

[35] A. Nedić and D. Bertsekas. Convergence rate of incremental subgradient algorithms. In *Stochastic optimization: algorithms and applications*, pages 223–264. Springer, 2001.

[36] D. Olteanu and M. Schleich. F: regression models over factorized views. *Proc. VLDB Endow.*, 9(13):1573–1576, 2016.

[37] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.

[38] L. Rdusseeun and P. Kaufman. Clustering by means of medoids. In *Proceedings of the statistical data analysis based on the L1 norm conference, neuchatel, switzerland*, volume 31, 1987.

[39] S. Rendle. Scaling factorization machines to relational data. *Proc. VLDB Endow.*, 6(5):337–348, 2013.

[40] Y. Roh, K. Lee, S. E. Whang, and C. Suh. Fairbatch: Batch selection for model fairness. In *ICLR 2021*. OpenReview.net, 2021.

[41] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD 2016*, pages 3–18. ACM, 2016.

[42] V. Shah, A. Kumar, and X. Zhu. Are key-foreign key joins safe to avoid when learning high-capacity classifiers? *Proc. VLDB Endow.*, 11(3):366–379, 2017.

[43] G. Strang. *Linear algebra and its applications*. Belmont, CA: Thomson, Brooks/Cole, 2006.

[44] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica. Neurocard: One cardinality estimator for all tables. *Proc. VLDB Endow.*, 14(1):61–73, 2020.

[45] B. Zhao and H. Bilen. Dataset condensation with differentiable siamese augmentation. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 12674–12685. PMLR, 2021.

[46] B. Zhao, K. R. Mopuri, and H. Bilen. Dataset condensation with gradient matching. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[47] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD 2018*, pages 1525–1539. ACM, 2018.