GoodCore: Data-effective and Data-efficient Machine Learning through Coreset Selection over Incomplete Data

CHENGLIANG CHAI, Beijing Institute of Technology, China

Given a dataset with incomplete data (e.g., missing values), training a machine learning model over the incomplete data requires two steps. First, it requires a data-effective step that cleans the data in order to improve the data quality (and the model quality on the cleaned data). Second, it requires a data-efficient step that selects a core subset of the data (called coreset) such that the trained models on the entire data and the coreset have similar model quality, in order to improve the training efficiency. The first-data-effective-then-data-efficient methods are too costly, because they are expensive to clean the whole data; while the first-data-efficient-then-data-effective methods have low model quality, because they cannot select high-quality coreset for incomplete data.

In this paper, we investigate the problem of coreset selection over incomplete data for data-effective and data-efficient machine learning. The essential challenge is how to model the incomplete data for selecting high-quality coreset. To this end, we propose the `GoodCore` framework towards selecting a good coreset over incomplete data with low cost. To model the unknown complete data, we utilize the combinations of possible repairs as possible worlds of the incomplete data. Based on possible worlds, `GoodCore` selects an expected optimal coreset through gradient approximation without training ML models. We formally define the expected optimal coreset selection problem, prove its NP-hardness, and propose a greedy algorithm with an approximation ratio. To make `GoodCore` more efficient, we further propose optimization methods that incorporate human-in-the-loop imputation or automatic imputation method into our framework. Experimental results show the effectiveness and efficiency of our framework with low cost.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Chengliang Chai. 2018. GoodCore: Data-effective and Data-efficient Machine Learning through Coreset Selection over Incomplete Data. *J. ACM* 37, 4, Article 111 (August 2018), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Data-effective machine learning (ML) (a.k.a. data-centric AI [**?**]) aims at obtaining high-quality training data to release the value of AI, because it is well-known that dirty data may severely degrade the performance of ML models [**? ?**].

Data-efficient ML focuses on making the training process more efficient. A commonly used strategy is to select a core subset of training data (or coreset) [**? ?**] to represent the entire dataset such that ML models trained on the coreset can achieve similar performance to the ML models trained on the entire dataset.

Author's address: Chengliang Chai, ccl@bit.edu.cn, Beijing Institute of Technology, China.

ACM 0004-5411/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

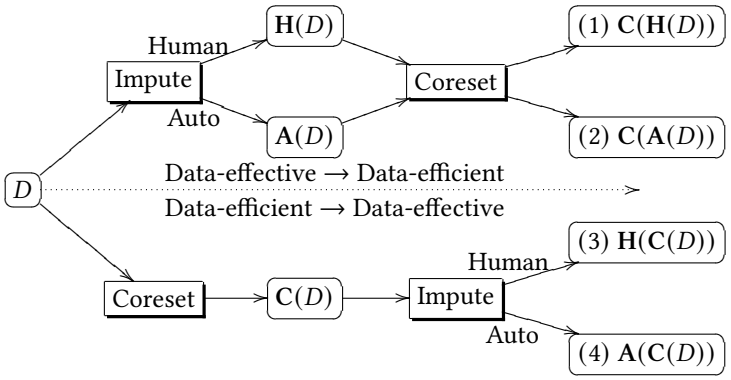

Fig. 1. Sequential methods.

Solution	Accuracy	Human Cost	Machine Cost
(1) $\mathbf{C}(\mathbf{H}(D))$	High	High	Low
(2) $\mathbf{C}(\mathbf{A}(D))$	Low	None	Low
(3) $\mathbf{H}(\mathbf{C}(D))$	Low	Low	Low
(4) $\mathbf{A}(\mathbf{C}(D))$	Low	None	Low
Our goal	High	None or Low	Low
(5) $\mathbf{H}(\mathbf{G}(D))$	High	Low	High
(6) $\mathbf{A}(\mathbf{G}(D))$	Medium	None	High
(7) $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$	High	Low	Low
(8) $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$	Medium	None	Low

Fig. 2. A comparison of different approaches (1–4: sequential methods; 5–8: our solutions).

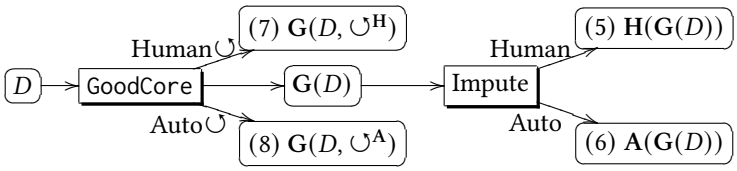

Fig. 3. Our proposal and its variants.

Apparently, users desire both data-effective ML (for training better ML models) and data-efficient ML (for saving training cost). In this work, our main goal is to support both data-effective and data-efficient ML over *incomplete data* where there are many missing values, which is very common in real-world scenarios [**? ? ?**].

Running data-effective and data-efficient tools sequentially. Intuitively, we can either run data imputation methods first for data-effective and then run coreset selection algorithms denoted by $\mathbf{C}(\cdot)$ for data-efficient, or vice versa. Moreover, for data-effective solutions through data cleaning, we generally consider two cases, either human-based solutions denoted by $\mathbf{H}(\cdot)$ or automatic solutions denoted by $\mathbf{A}(\cdot)$. In summary, we have the following four cases, as shown in Figure 1:

- *First data-effective (impute) then data-efficient (coreset):*
  (1) Impute-Human: $\mathbf{H}(D) \rightarrow$ Coreset: $\mathbf{C}(\mathbf{H}(D))$
  (2) Impute-Auto: $\mathbf{A}(D) \rightarrow$ Coreset: $\mathbf{C}(\mathbf{A}(D))$
- *First data-efficient (coreset) then data-effective (impute):*
  (3) Coreset: $\mathbf{C}(D) \rightarrow$ Impute-Human: $\mathbf{H}(\mathbf{C}(D))$

(4) Coreset: $\mathbf{C}(D) \rightarrow$ Auto-Human: $\mathbf{A}(\mathbf{C}(D))$

Next let's discuss the pros and cons of the above approaches.

Case (1) has high human cost, low machine cost, and high accuracy in terms of the trained ML models. Case (2) has zero human cost, low machine cost, but with low accuracy because automatic imputation may not be good enough. Case (3) has low human cost, low machine cost, but with low accuracy because corset selection over a dirty dataset may not ensure to compute a "good" coreset. Case (4) has no human cast, low machine cost, but with low accuracy with the similar reason as (3). The comparison of the above four methods can be found in Figure 2.

Our goal. Clearly, a primary goal is to achieve high accuracy for ML models, where only case (1) can achieve. Case (2) achieves low accuracy because automatic imputation is hard to be accurate. The main obstacle for making (1) practical is its high human cost. Hence, our *main goal* is to achieve high accuracy with no or low human cost, and with low machine cost.

Consider cases (3) and (4), the main reason for them to achieve low accuracy is because they cannot compute a good coreset directly from the dirty data. Intuitively, if we can compute a good coreset directly from the dirty data, we can cheaply clean the coreset to achieve high accuracy, where the "goodness" means that the subset of tuples selected from the dirty data is similar to the subset of tuples selected from the clean data.

Challenge. The main challenge of computing a good coreset from dirty data is to accurately estimate the ground truth of each missing value; otherwise, we cannot select a coreset to well represent the clean data. This is a known hard problem because each missing value may have multiple possible repairs. Also, because a coreset selection algorithm is typically iterative that each tuple is selected per iteration [**?**], selecting a bad tuple may cause cascade amplification to the following iterations, resulting in a bad coreset.

Our proposal. To tackle the above challenge, we model the combinations of possible repairs as possible worlds of the original dirty data D. We then formulate it as an optimization problem for selecting an expected optimal coreset that can represent the possible worlds of D via gradient approximation without training in advance. We prove this problem to be NP-hard. We propose an approximate algorithm, called `GoodCore`, denoted by $\mathbf{G}(\cdot)$, with the main idea to iteratively add a tuple with the highest utility into the coreset. After a good coreset is computed, we can either use human imputation or automatic imputation to impute the data, as shown in Figure 3. We further elaborate these two methods below:

(5) `GoodCore`: $\mathbf{G}(D) \rightarrow$ Impute-Human: $\mathbf{H}(\mathbf{G}(D))$
(6) `GoodCore`: $\mathbf{G}(D) \rightarrow$ Impute-Auto: $\mathbf{A}(\mathbf{G}(D))$

However, one main drawback is that modeling possible worlds of D is computationally expensive, which hinders the practicability of the `GoodCore` algorithm. To address this high computational cost problem, we further propose **optimization** methods to integrate imputation-in-the-loop (with either humans or automatic methods) into the `GoodCore` algorithm (see methods 7 and 8 in Figure 3). To this end, the optimized algorithms can significantly reduce the number of possible worlds, thus achieving low computational cost.

(7) `GoodCore` with human-in-the-loop imputation: $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$
(8) `GoodCore` with machine-in-the-loop imputation: $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$

A comparison of methods (5)–(8) is given in Figure 2. Note that method (7) is the best solution because it can achieve a high ML accuracy with low human cost and low machine cost.

Contributions We make the following contributions.

(i) *Two birds with one stone.* We study the problem of solving both data-effective and data-efficient ML in one framework, which is an important but not addressed problem. (Section 3)
(ii) *NP-hardness and approximate solutions.* We prove the NP-hardness of the problem. We propose a greedy algorithm with an approximate ratio. (Section 4)
(iii) *Optimizations.* We develop optimization techniques that integrate imputation-in-the-loop into the coreset selection process, to improve the efficiency while achieving high accuracy. We also analyze the convergence rate of our method and theoretically prove that it can converge fast. (Section 5)
(iv) *Experiments.* We conduct extensive experiment on 6 real-world datasets and compare with 10 baselines to show that `GoodCore` can select a well-performed coreset to achieve both data-effective and data-efficient ML while consuming a low human cost. (Section 6)

2 BACKGROUND OF CORESET SELECTION

In this section, we introduce the background of coreset selection on complete data, denoted by D_c.

2.1 Gradient Descent for Machine Learning

Gradient descent [**?**] is the most typical optimization algorithm to train ML models. At a high level, it tweaks the parameters iteratively to minimize a given convex and differentiable function to its local minimum.

Let $D_c = \{t_1, t_2, ..., t_n\}$ be a set of train tuples (without missing values), where $t_i = (\mathbf{x}_i, \mathbf{y}_i)$, $\mathbf{x}_i \in \mathbb{R}^d$ denotes the vector of features and $\mathbf{y}_i$ denotes the corresponding label. The goal of training on D_c is to find the best parameter θ^* of an model by minimizing the loss:

$$\theta^* = \arg\min_{\theta \in \vartheta} f(\theta), f(\theta) = \frac{1}{n} \sum_{i=1}^{n} f_i(\theta, t_i) \tag{1}$$

where ϑ is the parameter space. For ease of representation, we abbreviate $f_i(\theta, t_i)$ as $f_i(\theta)$ to represent the loss of the i-th train example. Generally speaking, the gradient descent approach is always applied to find the minimizer of Eq. 1, where the **full gradient** (sum of the gradients over all training tuples), denoted by $\nabla f(\theta) = \sum_{i=1}^{n} \nabla f_i(\theta)$, has to be computed iteratively.

Besides incremental gradient methods like stochastic gradient descent (SGD) that can be leveraged to accelerate the iterative gradient computation, there are other popular and orthogonal methods, such as coreset, which will be discussed next.

2.2 Coreset over Complete Data

Coreset. To make training more efficient, instead of learning from entire D_c, one research question is that whether we can compute a small subset $\mathbf{C}(D_c)$ of D_c such that learning with $\mathbf{C}(D_c)$ can hopefully achieve the same performance as learning with D_c. This small selected subset is called **coreset** [**? ?**]. In the following, we simply write $\mathbf{C}(D_c)$ as C when it is clear from the context.

The state-of-the-art coreset selection solutions are mostly based on gradient approximation [**? ?**]. Suppose that θ denotes the parameter of an ML model trained over the full dataset, and θ' denotes the parameter of the same model trained over the coreset. Intuitively, the objective of gradient approximation for coreset selection is to make $\nabla f(\theta')$ as close as possible to $\nabla f(\theta)$. To this end, existing solutions focus on *selecting the coreset that minimizes the upper bound of gradient approximation error* ($\|\nabla f(\theta) - \nabla f(\theta')\|$). Next, let's formally define it from scratch.

Gradient-based coreset selection is to minimize the **gradient approximation error (GA error)** between the full gradient w.r.t. D_c and the weighted sum of gradients w.r.t. the coreset C (or coreset gradient). Formally, Eq. 2 tries to minimize the GA error by considering all possible

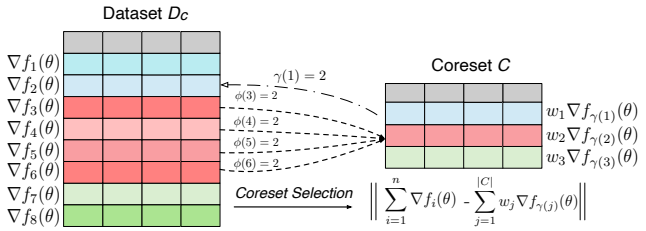

Fig. 4. Example of coreset selection.

parameters $\theta \in \vartheta$ (*i.e.*, $\max_{\theta \in \vartheta}$), where "$\|\cdot\|$" denotes the normed difference. Next, we introduce the coreset gradient.

$$C^* = \arg\min_{C \subseteq D_c, w_j \geq 0} \max_{\theta \in \vartheta} \| \underbrace{\sum_{i=1}^{n} \nabla f_i(\theta)}_{\text{full gradient}} - \underbrace{\sum_{j=1}^{|C|} w_j \nabla f_{\gamma(j)}(\theta)}_{\text{coreset gradient}} \|, \quad \underbrace{\qquad\qquad}_{\textbf{gradient approximation error}} \tag{2}$$
$$s.t.\ |C| \leq K$$

Because the coreset is a subset of the complete dataset (*i.e.*, $C \subseteq D_c$), we use $\gamma(j) = i$ (where $j \in [1, |C|], i \in [1, n]$) to denote that the j-th tuple in C (denoted by c_j) is the i-th tuple in D_c, *i.e.*, t_i. In other words, γ is an index mapping from C to D_c.

Recall that the key idea of the coreset is to *use a subset of tuples to represent the entire set*. Eq. 2 potentially contains another important mapping ϕ from D_c to C to indicate this, *i.e.*, $\phi(i) = j, i \in [1, n], j \in [1, |C|]$, which is highly related to the weight. Specifically, let $\phi(i) = j$ denote that we will assign t_i to c_j (use c_j to represent t_i) and use $\nabla f_{\gamma(j)}$ to represent ∇f_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 D_c. Based on ϕ, w_j is defined as the weight of the c_j, which is the number of tuples in D_c assigned to the c_j, *i.e.*, $w_j = |\{t_i | \phi(i) = j, i \in [1, n]\}|$ (c_j is utilized to represent w_j tuples in D_c).

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

Example 1. *Let's consider a case of the gradients of each tuple, as shown in Figure 4. Suppose that for any θ, $\nabla f_1(\theta) \approx \nabla f_2(\theta)$, $\nabla f_3(\theta) \approx \nabla f_4(\theta) \approx \nabla f_5(\theta) \approx \nabla f_6(\theta)$ and $\nabla f_7(\theta) \approx \nabla f_8(\theta)$. In this case, based on Eq. 2, if one wants to find an optimal coreset with a size of 3, i.e., $K = 3$, the solution can be $C^* = \{t_2, t_5, t_7\}$ ($\gamma(1) = 2, \gamma(2) = 5$ and $\gamma(3) = 7$), associated with $w_1 = 2, w_2 = 4, w_3 = 2$ because $\phi(1) = \phi(2) = 1, \phi(3) = \phi(4) = \phi(5) = \phi(6) = 2$ and $\phi(7) = \phi(8) = 3$. In this way, C^* can be one of the optimal coresets that can well approximate the full gradient because $\| \sum_{i=1}^{8} \nabla f_i(\theta) - \sum_{j=1}^{3} w_j \nabla f_{\gamma(j)}(\theta) \|$ is minimized, which is close to 0.*

Key observation. We can observe from Example 1 that in order to minimize the GA error, we should set $\phi(i) = j$, where ∇f_i and $\nabla f_{\gamma(j)}$ are likely to be close. Therefore, computing the coreset is similar to computing the K exemplars [**?**] of the gradients, if all the gradients of tuples can be computed.

Upper bound minimization of GA error. We can see from Eq. 2 that to solve the equation, the gradients have to be computed, which have a close relationship with the parameter θ. However, the main bottleneck is that the entire parameter space ϑ is too expensive to explore. Hence, a

typical solution is to first compute the upper bound of GA error (Eq. 3), then generalize [**? ? ?**] the upper bound computation to the entire parameter space (Eq. 4), and finally select the coreset to minimize the bound. To be specific, using the triangle equation, for any particular θ, we have:

$$\|\sum_{i=1}^{n}\nabla f_i(\theta) - \sum_{j=1}^{|C|} w_j \nabla f_{\gamma(j)}(\theta)\| \leq \sum_{i=1}^{n}\|\nabla f_i(\theta) - \nabla f_{\gamma(\phi_\theta(i))}(\theta)\| \tag{3}$$

Together with the aforementioned observation, given a coreset C, the upper bound is minimized when ϕ assigns every tuple t_i to the tuple in C with most gradient similarity, *i.e.*, $\|\sum_{i=1}^{n}\nabla f_i(\theta) - \sum_{j=1}^{|C|} w_j \nabla f_{\gamma(j)}(\theta)\| \leq \sum_{i=1}^{n}\min_{c_j\in C}\|\nabla f_i(\theta) - \nabla f_{\gamma(j)}(\theta)\|$.

For the entire space ϑ, it has been proved in recent works [**? ? ?**] that for convex ML problems (corresponding to an optimization problem in which the objective function is a convex function), the normed gradient difference between tuples can be efficiently bounded by:

$$\forall i, j, \max_{\theta\in\vartheta}\|\nabla f_i(\theta) - \nabla f_j(\theta)\| \leq \max_{\theta\in\vartheta} O(\|\theta\|)\cdot\|\mathbf{x}_i - \mathbf{x}_j\| \tag{4}$$

where $\|\mathbf{x}_i - \mathbf{x}_j\|$ is the Euclidean distance between feature vectors of two tuples, namely *feature distance*, and $O(\|\theta\|)$ is a constant. Hence, we can conclude that **GA error can be bounded independent of the optimization problem in practice,** *i.e.*, **any particular** θ. Finally, considering Eq. 3 and Eq. 4 together, *the coreset selection problem can be converted to:*

$$C^* = \arg\min_{C\subseteq D_c}\sum_{i=1}^{n}\min_{c_j\in C} s_{ij},\ \text{s.t.}\ |C| \leq K \tag{5}$$

where $s_{ij} = \|\mathbf{x}_i - \mathbf{x}_{\gamma(j)}\|$ for ease of representation. The above equation indicates that given a train data D_c and a coreset C, we use $S = \sum_{i=1}^{n}\min_{c_j\in C} s_{ij}$ to score the coreset. The lower the score, the smaller upper bound of the GA error we can get, which indicates a better coreset. To summarize, solving Eq. 5 is to minimize the upper bound of the GA error (*i.e.*, select the coreset with the lowest score) by just considering the feature vectors of the training tuples without training in advance.

Note that Eq. 4 holds for tuples associated with the same label [**? ?**]. Therefore, in practice, we respectively select coresets for tuples with different labels and combine them. Suppose that we aim to select a coreset with size K for a binary classification task (label 1: 60%, label 0: 40%), so we select a coreset with size $60\%K$ for tuples with label 1 and another one with $40\%K$ for tuples with label 0.

Our scope. In this paper, we focus on the convex problems (*e.g.*, logistic regression, support vector machine, etc.) because for such problems the gradient difference can be well bounded by the difference between feature vectors. Note that, for other ML algorithms such as deep neural networks, they can also be trained using selected coreset to achieve good training accuracy (see Section 6 for our experimental findings).

3 CORESET OVER INCOMPLETE DATA

In this section, we will formally define the problem of coreset selection over incomplete data (Section 3.1) and then describe our proposed framework to solve the problem (Section 3.2).

3.1 Problem Definition

As discussed above, we have to compute the coreset score S, so as to produce a good coreset. To this end, the feature distances can be computed as a pre-processing step, based on which the coreset

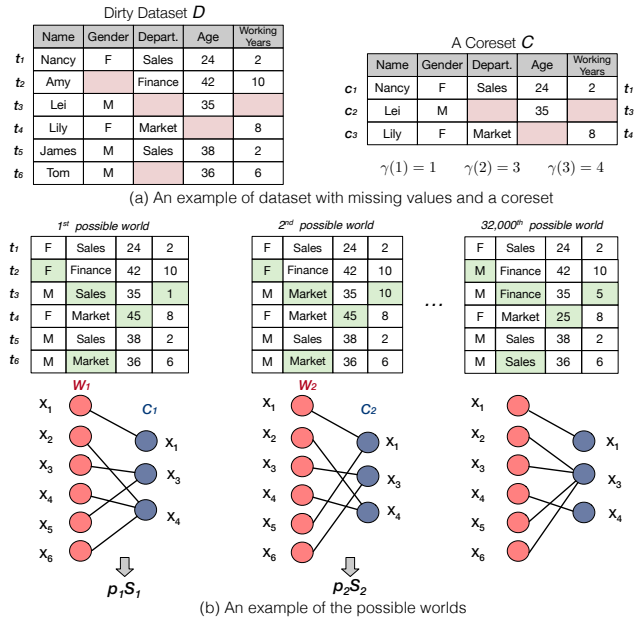

Fig. 5. Example of coreset selection with missing values

.

score can be computed. However, when there exists incomplete data with missing values, even the feature distances are hard to compute accurately, let alone selecting a proper coreset.

Incomplete data. Formally, suppose that D has M attributes, denoted by $\{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_M\}$. Each attribute $\mathcal{A}_m, m \in [1, M]$ represents a domain set including the `Null`, (*i.e.*, $\texttt{Null} \in \mathcal{A}_m$), in which each tuple in D can take value on this attribute. $|\mathcal{A}_m|$ denotes the domain size. Then, each tuple $t_i \in \mathcal{A}_1 \times \mathcal{A}_2 \times, ..., \times \mathcal{A}_m$. Let $t_i[m]$ denote the value of the $m-$th attribute of t_i, *i.e.*, $t_i[m] \in \mathcal{A}_m$.

For a tuple $t_i \in D$, if $\exists\ t_i[m] = \texttt{Null}, m \in [1, M]$, t_i is an incomplete tuple, denoted by $\mathbb{I}[t_i] = 1$, otherwise $\mathbb{I}[t_i] = 0$. Let us better illustrate this using an example.

Example 2. *As shown in Figure 5(a), there are 6 tuples in the table D with five attributes (an excerpt from a large table). For example, $\mathcal{A}_2$ is the* `Gender` *attribute, i.e., $\mathcal{A}_2 = \{\texttt{M}, \texttt{F}, \texttt{Null}\}$. Among these tuples, t_2, t_3, t_4, t_6 have missing values, e.g., $\mathbb{I}[t_2] = 1, \mathbb{I}[t_1] = 0$. Given a coreset as shown on the right side, if there are no missing values, we can assign each tuple $t_i \in D$ to its most similar tuple in C (compute $\min_{c_j \in C} s_{ij}$), and then sum these feature distances up to compute the coreset score S. However, given these missing values, the feature distances cannot be computed accurately (e.g., s_{12}, s_{13}, s_{22}, etc.), and thus the assignment of tuples in D cannot be determined precisely. Hence, the coreset score is not precise, and thereby leads to a coreset that cannot well represent the full complete (clean) data.*

As discussed above, *imputation before coreset selection* suffers from either large cost (human imputation) or large number of possible repairs (automatic imputation), while *imputation after coreset selection* cannot obtain a good coreset because of the inaccurate feature distance computation (see Example 2).

Therefore, an essential problem is to select a good coreset that can represent the complete dataset D_c, which relies on accurate coreset score computation given D_c that is the unknown ground truth.

Fortunately, the possible repairs of D can be modeled by possible worlds [? ? ? ?], based on which we can effectively select the coreset over incomplete data.

Possible worlds. Given the incomplete dataset D, $\forall t \in D$ and $\mathbb{I}[t] = 1$, $\forall t[m] = \texttt{Null}, m \in [1, M]$, we assign a value in $\mathcal{A}_m \setminus \{\texttt{Null}\}$ to $t[m]$ as an imputation (a.k.a. a possible repair). Thus, we have an assignment for all the missing values in D, which corresponds to a possible world W. Since there exist a large number of possible assignments, we define the set of possible worlds as $\mathcal{I}_\mathcal{W} = \{W_k | k \in [1, |\mathcal{I}_\mathcal{W}|]\}$.

Let us better illustrate this using an example.

Example 3. *Given D, for tuples t_2, t_3, t_4, t_6 with missing values, we have a large number of possible assignment as shown in Figure. 5(b), each of which corresponds to a possible world (we omit the* `Name` *attribute because there is no missing value on this attribute). Suppose that there are 2 (4/100/10) types of values of the attribute* `Gender` *(*`Department`*/*`Age`*/*`Working years`*), there exist 32,000 possible worlds in total.*

Note that for numerical attributes, we will bin them into different buckets, such that we can treat them as categorical values and avoid the unlimited number of possible worlds.

Even with possible worlds, the score computation of coreset remains challenging. Each possible world of D is a complete dataset, and thus given a coreset, the score can be directly computed considering the feature distances, as discussed in Section 2.2. However, the crucial issue is that each possible world could be the ground truth, *i.e.*, D_c, but each one leads to a different score.

Example 4. *As shown in Figure. 5(b), the two possible worlds W_1 and W_2 are only different in t_3, leading to a different feature vector $\mathbf{x}_3$, which makes the score computation a difference. To be specific, given the same coreset C with tuples $t_1(c_1)$, $t_3(c_2)$ and $t_4(c_3)$, because of a different $\mathbf{x}_3$, the closest feature distance of $\mathbf{x}_5$ in W_2 becomes $\mathbf{x}_1$, rather than $\mathbf{x}_3$ in W_1. And the closest feature distance of $\mathbf{x}_6$ in W_2 becomes $\mathbf{x}_3$, rather than $\mathbf{x}_4$ in W_1. Therefore, the coreset scores, i.e., the sum of these closest feature distances of tuples are different among possible worlds.*

Example 4 shows that different possible worlds make the mapping ϕ different, which leads to different scores. Hence, to get a good coreset without the ground truth, an intuitive solution is to compute the expected coreset score considering all possible worlds. By doing so, although we cannot get the complete data (D_c) in advance, we can focus on how to select an informative coreset that can represent the possible worlds of D.

Next, we formally define the studied problem.

Expected optimal coreset selection over incomplete data. Given D, we have a number of possible worlds $\mathcal{I}_\mathcal{W} = \{W_k\}$. Then given a subset (coreset) $C \subset D$, for different W_k, we have the corresponding C_k with the same tuples as C but probably different imputations. For C_k, we can compute a score $S_k = \sum_{i=1}^{n} \min_{c_j \in C_k} s_{ij}$, where $s_{ij} = \|\mathbf{x}_i - \mathbf{x}_{\gamma(j)}\|$ and both feature vectors are from $\{W_k\}$. Then, we have the expectation $\mathrm{E}[C] = \sum_{k=1}^{|\mathcal{I}_W|} p_k S_k$, where p_k denotes the probability of the appearance of $\{W_k\}$. Finally, our problem becomes how to compute the coreset C with the lowest expectation of GA error upper-bound. Formally, we have

$$C^* = \arg\min_{C \subseteq D} \mathrm{E}[C], \text{ s.t. } |C| \leq K \tag{6}$$

For example, given D, the corresponding possible worlds and a coreset C in Figure 5, we have different C_k with the same tuples (containing t_1, t_3, t_4) but probably different imputations. For each C_k, we will compute S_k, and finally compute $\mathrm{E}[C]$. Solving Eq. 6 can result in an informative coreset with incomplete tuples being selected. After these tuples imputed by a human, *i.e.*, Case (5), or state-of-the-art automatic method, *i.e.*, Case (6), we can derive a good coreset.

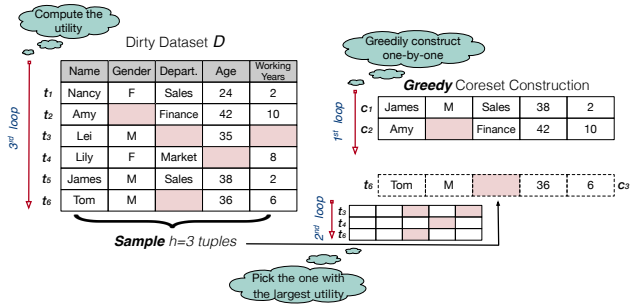

Fig. 6. The GoodCore framework.

Algorithm 1: GoodCore Framework

Input: Incomplete train data D, coreset size K, sample size h.
Output: A coreset $C \subseteq D$, weight $\mathbb{W} = \{w_j\}$,$|C| = |\mathbb{W}| = K$.

1 $C = \emptyset$;
2 **while** $|C| < K$ **do**
3 /*1st loop*/
4 Sample h tuples as $T_{sample} \subseteq D \setminus C$
5 **for** *each tuple* $t \in T_{sample}$ **do**
6 /*2nd loop*/
7 $\mathrm{E}[t|C]$ = `ComputeUtility`(t, C, D); /*3rd loop*/
8 $t^* = \arg\max_{t \in T_{sample}} \mathrm{E}[t|C]$;
9 $C = C \cup \{t^*\}$;
10 **for** $t \in C$ **do**
11 **if** $\mathbb{I}[t] = 1$ **then**
12 Impute t by a human or automatic method.
13 **for** $j = 1$ *to* $|C|$ **do**
14 **for** $i = 1$ *to* n **do**
15 **if** $c_j = \arg\min_{c_{j'} \in C} \max_{\theta \in \vartheta} \|\nabla f_i(\theta) - \nabla f_{\gamma(j')}(\theta)\|$ **then**
16 w_j += 1;
17 **return** $C, \mathbb{W}$;

3.2 Goodcore Framework

Next, we will introduce our proposed GoodCore framework to solve Eq. 6, which is non-trivial because it is NP-hard. But fortunately, we prove that it has the sub-modular property (see Section 4). Hence, GoodCore uses a greedy framework with three loops to solve the problem with an approximate ratio.

At a high level, the greedy strategy adds one tuple with the largest "utility" to the coreset iteratively, which can be considered as the first loop. In each iteration, we have to iterate tuples in D to select the one with the largest utility, which is the second loop. Naturally, we have to compute the utility of each tuple, where all tuples in D have to be considered, leading to the third loop.

Next, we will further illustrate the framework using Figure 6 and Algorithm 1.

The first loop (lines 2-9) of the greedy algorithm is to add the tuple t^* with the maximum *utility* (*i.e.*, $\mathrm{E}[t|C] = \mathrm{E}[C] - \mathrm{E}[C \cup \{t\}]$) into the coreset iteratively for K times. To be specific, the "utility" of a tuple t denotes the reduction of expectation of GA error after adding t into the coreset C.

Suppose that $K = 3$. Figure 6 (the 1^{st} loop part) shows the situation that there already have been 2 tuples in C, and we are going to add the third tuple into the coreset.

The second loop (lines 5-7) computes the utilities of tuples that are not in coreset C, based on which the best one is picked for the first loop. An ideal solution is to consider all tuples in $D \setminus C$, which is prohibitively expensive, so in practice we use an efficient method to accelerate this loop by uniformly sampling h tuples as T_{sample} (line 4) and then selecting the best one from T_{sample} (line 8). The difference is that theoretically, considering all tuples has an approximate ratio 1-$\frac{1}{e}$ (because of the sub-modular property), while the sampling method holds a (1-$\frac{1}{e} - \epsilon$) ratio [**?**], where ϵ is related to the sampling ratio.

As shown in Figure 6, suppose that $h = 3$, and we sample $\{t_3, t_4, t_6\}$ from $\{t_1, t_3, t_4, t_6\}$. Then the second loop iterates the three tuples and computes the utility for each one (the third loop).

The third loop (line 7) will loop through all tuples in D, so as to compute the utility of tuple t used in the second loop. To be specific, the core part of the utility computation (*i.e.*, `ComputeUtility`) is to compute $\mathrm{E}[C] = \sum_{k=1}^{|\mathcal{I}_W|} p_k S_k = \sum_{k=1}^{|\mathcal{I}_W|} p_k (\sum_{i=1}^{n} \min_{c_j \in C_k} s_{ij})$, from which we can see that it is inevitable to iterate the n tuples in D. However, the most challenging part is that we also have to enumerate a large number of possible worlds. We will illustrate how to solve this in details in Section 4.

The imputation step (line 12). After `GoodCore` selects the coreset C using the above 3 loops, we can leverage a human or automatic method to impute the tuples that are incomplete in C, which correspond to Case (5) and Case (6) in Section 1 respectively.

Weights computation (lines 13-16). It computes the weight of each tuple in C, which will be used to approximate the full gradient during training. For training, tuples in the coreset are randomly shuffled. Afterwards, suppose that in each step of the gradient decent, when we use $c_j \in C$ to update the gradient, we compute the gradient (∇f_j) of c_j first, and then use $w_j \nabla f_j$ to update the model parameters. w_j is the number of tuples in D assigned to c_j. The above steps repeat until the model converges.

Optimization. Unfortunately, the 3-loop computation of the strategy is rather expensive due to the large number of possible worlds (Section 4). To address this, we can integrate either human-in-the-loop or the automatic method into `GoodCore` framework (Section 5). It iteratively imputes one incomplete tuple or a mini-batch of incomplete tuples. Once the tuple(s) is (are) computed and added to the coreset within the first loop, the number of possible worlds can be significantly reduced, and so does the computational cost. In addition, how to select an appropriate coreset size K is an important problem, which will be discussed in Section 6.3.

4 GOODCORE ALGORITHM

In this section, we will illustrate `GoodCore` algorithm in details for solving Eq. 6, which is proven to be prohibitively expensive (Section 4.1). Then we focus on how to compute the expectation using possible worlds (Section 4.2) in the algorithm.

4.1 Problem Complexity

Let us first discuss the time complexity of finding the optimum of Eq. 6.

Theorem 1. *The problem of expected optimal coreset selection over incomplete data is NP-hard.*

Proof. Let us consider a special case that there is no missing value in D. Our problem becomes the typical coreset selection problem over complete data, which has been proven to be NP-hard

by reduction from the Minimum Vertex Cover problem [? ? ?]. Hence, our problem is also NP-hard. □

Theorem 2. *The problem of expected optimal coreset selection over incomplete data has the sub-modular property.*

Proof. First, we regard $\mathrm{E}[C] = \sum_{k=1}^{|\mathcal{I}_W|} p_k S_k$ as a utility function, where $S_k = \sum_{i=1}^{n} \min_{c_j \in C_k} s_{ij}$. In fact, S_k can be regarded as a function of the coreset score computation over complete data, which has already proven to have the sub-modular property [? ? ?]. Therefore, consider the property that a non-negative linear combination of sub-modular functions is also sub-modular [?]. To be specific, given any sub-modular function $g_1, g_2, \ldots, g_k$ and non-negative numbers $\alpha_1, \alpha_2, \ldots, \alpha_k$. Then the function $\mathcal{G}$ defined by $\mathcal{G} = \sum_{i=1}^{k} \alpha_i g_i$ is sub-modular. Hence, we can conclude that our studied problem is a sub-modular problem because $\mathrm{E}[C] = \sum_{k=1}^{|\mathcal{I}_W|} p_k S_k$, where $p_k > 0$. □

The greedy algorithm. Given the sub-modular property, naturally, we can design a greedy algorithm with an approximate ratio. As shown in Algorithm 1, we greedily add one tuple to the coreset at each iteration. The added tuple should have the largest utility computed by $\mathrm{E}[t|C] = \mathrm{E}[C] - \mathrm{E}[C \cup \{t\}]$. Hence, the key component is that given the original train data (D) and a coreset (C or $C \cup \{t\}$), how to compute the expectation of GA error ($\mathrm{E}[C]$ or $\mathrm{E}[C \cup \{t\}]$) of the coreset. However, it is non-trivial because of the large number of possible worlds. We will first introduce how to compute the probability p_k, and describe the expectation computation in Section 4.2. After K tuples are added, we can impute missing tuples in the coreset generated by `GoodCore`.

4.2 Expectation Computation

Possible world probability. To compute the expectation, it is inevitable to derive the probability of each possible world, which can be taken as a pre-processing step in our framework. To be specific, since tuples with missing values are always imputed independently [?], given a possible world W_k, the probability p_k can be computed by $p_k = \prod_{t \in W_k, \mathbb{I}[t]=1} p_k^t$, where p_k^t denotes the probability of the appearance of tuple t with $\mathbb{I}[t] = 1$. Besides, apparently $p_k^t = 1$ when $\mathbb{I}[t] = 0$, so $p_k = 1$ if there are only complete tuples. Therefore, our focus is on how to get the value of p_k^t, which can be solved by many approaches, like statistic methods and learning-based methods (see [?] for a survey). In this paper, we use the learning-based method [?] with a Python library [?] to generate the probability, which can be easily replaced by other libraries or domain-specific methods. During training, learning-based methods take D as input and learn a model $\mathcal{M}$ to describe the joint data distribution. For inference, we have $P(\mathcal{A}_i|\mathbf{x}, v_{mask}) = \mathcal{M}(\mathbf{x}, v_{mask}, \omega^*)$, where the model takes as input the feature vector $\mathbf{x}$ of t, the mask vector v_{mask} (indicating which attributes are missing) and the model parameter ω^*, outputs the probability distribution of a missing attribute $\mathcal{A}_i$.

Suppose that t just has one missing attribute $\mathcal{A}_i$, and then v_{mask} is a one-hot vector with $v_{mask}[i] = 0$. Hence, we can directly obtain p_k^t from the distribution $P(\mathcal{A}_i|\mathbf{x}, v_{mask})$. For t with multiple missing attributes, we can also compute p_k^t using the chain rule. If t has two missing values of $\mathcal{A}_i$ and $\mathcal{A}_j$, to compute p_k^t, we have to compute $P(\mathcal{A}_i, \mathcal{A}_j|\mathbf{x}, v_{mask})$, abbreviated as $P(\mathcal{A}_i, \mathcal{A}_j) = P(\mathcal{A}_i)P(\mathcal{A}_j|\mathcal{A}_i)$. $P(\mathcal{A}_i)$ can be obtained by masking the i-th and j-th attribute in v_{mask}. Then, we only mask the j-th attribute and impute different values of $\mathcal{A}_i$ to obtain $P(\mathcal{A}_j|\mathcal{A}_i)$.

Example 5. *In Figure 5(a), suppose that for the first possible world, we have to compute* $p_1 = p_1^2 \times p_1^3 \times p_1^4 \times p_1^6$. *For instance, to compute* p_1^3, *given the trained deep learning model, we feed* $\{\texttt{Lei}, \texttt{M}, \texttt{Mask}, 35, \texttt{Mask}\}$ *and a one-hot vector* $\{1, 1, 0, 1, 0\}$ *into the model and compute the probability distribution of this tuple, from which we can get* p_1^3, *i.e., the probability of* $\{\texttt{Lei}, \texttt{M}, \texttt{Sales}, 35, 1\}$.

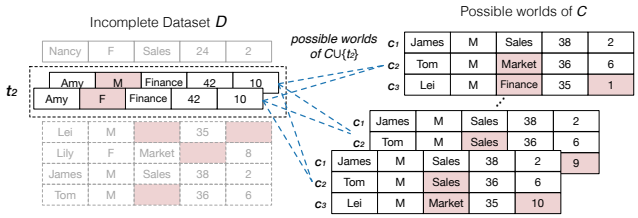

Fig. 7. Tuple-based expectation computation.

Compared with statistical approaches, deep learning-based methods use more powerful models with good learning capacity and consider the correlation between attributes. For practitioners, they can use any ad-hoc method to compute the probability.

Brute-force expectation computation. Recap that $\mathrm{E}[C] = \sum_{k=1}^{|\mathcal{I}_W|} p_k(\sum_{i=1}^{n} \min_{c_j \in C_k} s_{ij})$. Intuitively, the brute-force method is to enumerate each possible world, compute the probability and finally get the expectation. However, there are a huge number of possible worlds, which makes the computation prohibitively expensive. Specifically, we assume the attribute number M and $|\mathcal{A}_m|, m \in [1, M]$ are constants, so the number of possible worlds of each tuple is a constant, denoted by L. Suppose that the number of tuples with missing values is $O(n)$, so the number of possible worlds ($|\mathcal{I}_W|$) is $O(L^n)$. Given a coreset C, the time complexity to compute $\mathrm{E}[C]$ is $O(nL^n)$, which is rather expensive.

Tuple-based expectation computation. To further elaborate, we can easily expand $\mathrm{E}[C]$ as follows:

$$\begin{aligned}\mathrm{E}[C] =& \underbrace{p_1(\min_{c_j \in C_1} s_{1j} + \min_{c_j \in C_1} s_{2j} + \cdots + \min_{c_j \in C_1} s_{nj})}\\ &+ \underbrace{p_2(\min_{c_j \in C_2} s_{1j} + \min_{c_j \in C_2} s_{2j} + \cdots + \min_{c_j \in C_2} s_{nj})} + \cdots\\ &+ \underbrace{p_{|\mathcal{I}_W|}(\min_{c_j \in C_{|\mathcal{I}_W|}} s_{1j} + \min_{c_j \in C_{|\mathcal{I}_W|}} s_{2j} + \cdots + \min_{c_j \in C_{|\mathcal{I}_W|}} s_{nj})}.\end{aligned}$$

We can see from the above equation that these underlined terms are only related to $t_1 \in D$ as well as $\{C_1, C_2, \cdots, C_{|\mathcal{I}_W|}\}$, *i.e.*, the coresets corresponding to the $|\mathcal{I}_W|$ possible worlds. However, as the coreset C is much smaller than the full data D, the number of possible worlds of C will be also much smaller than $|\mathcal{I}_W|$, and thus there will be many duplicates among $\{C_1, C_2, \cdots, C_{|\mathcal{I}_W|}\}$. Therefore, many of these underlined terms have identical variable parts, *i.e.*, $\min_{c_j \in C_k} s_{1j}$, when they are associated with the same C_k. These terms are *like terms*. Combining these like terms (*i.e.*, $\sum_{k=1}^{|\mathcal{I}_W|} p_k \min_{c_j \in C_k} s_{1j}$), we can get the expectation of $\min_{c_j \in C} s_{1j}$, denoted by $\mathrm{E}[\min_{c_j \in C} s_{1j}]$.

In short, we can convert the expectation computation over the possible worlds of the entire training set D to the sum of expectation of each tuple in D, as follows:

$$\mathrm{E}[C] = \sum_{k=1}^{|\mathcal{I}_W|} p_k(\sum_{i=1}^{n} \min_{c_j \in C_k} s_{ij}) = \sum_{i=1}^{n} \mathrm{E}[\min_{c_j \in C} s_{ij}] \tag{7}$$

Example 6. *Figure 7 shows how to compute* $\mathrm{E}[\min_{c_j \in C} s_{2j}]$. *Instead of enumerating* $|\mathcal{I}_W|$ *possible worlds by the brute-force method, we can enumerate a much smaller number of possible worlds of* $C \cup t_2$, *compute the corresponding probabilities and finally get the tuple expectation. Specifically, The*

Algorithm 2: `ComputeUtility` (3rd-loop to compute $\mathrm{E}[t|C]$)

Input: Incomplete train data D, current coreset C, a sampled tuple t.
Output: The expectation $\mathrm{E}[t|C]$.

```
1  Ĉ = C ∪ {t};
2  E[Ĉ] = 0;
3  for each tuple t_i ∈ D do
4      if 𝕀[t] = 0 and 𝕀[t_i] = 0 then
5          E[Ĉ] += min_{c_j ∈ Ĉ} s_ij;
6      else
7          Get the possible worlds of Ĉ ∪ {t_i};
8          Compute E[min_{c_j ∈ Ĉ} s_ij] using these possible worlds and their probabilities;
9          E[Ĉ] += E[min_{c_j ∈ Ĉ} s_ij];
10 E[t|C] = E[C] − E[Ĉ];
11 return E[t|C];
```

left part of Figure 7 shows the possible worlds of the tuple, the right part shows the possible worlds of the coreset, and their combination is the possible worlds of $C \cup t_2$. Then, following Eq. 7, we can iterate the tuples in D, compute their expectations and sum them up to derive $\mathrm{E}[C]$.

Time complexity. Since the coreset size is K, and the number of tuples with missing values in the coreset is $O(K)$, the time complexity of computing $\mathrm{E}[C]$ using tuple-based method is $O(nL^K)$, where K is much smaller than n, compared with the brute-force method. However, note that computing $\mathrm{E}[C]$ is just the third loop in the entire framework. Besides, the first two loops incrementally add K tuples into the coreset, and sample h tuples for tuple selection respectively. Hence, the overall time complexity of coreset selection over incomplete data is $O(KhnL^K)$, which is still expensive when K is not small enough. In the next section, we involve the imputation-in-the-loop strategies to achieve further improvement.

5 OPTIMIZED GOODCORE WITH IMPUTATION-IN-THE-LOOP

As discussed above, it is rather expensive to directly compute all the K tuples in the coreset. Hence, in this section, we propose to involve the imputation-in-the-loop mechanism that asks the human, *i.e.*, Case (7), or automatic method, *i.e.*, Case (8) to impute these missing values iteratively while they are generated by Algorithm 1.

The advantages of this optimization are two-fold. First, with more and more missing values being imputed, the number of possible worlds is greatly reduced, which reduces the machine cost a lot. Second, for human-in-the-loop imputation, it allows us to gradually impute the tuples accurately, and thus the coreset score computation can be more and more accurate, which produces a better coreset.

5.1 One Tuple Each Iteration

In fact, we can just slightly modify Algorithm 1 to achieve the imputation-in-the-loop strategy. To be specific, in the first loop, we will iteratively impute the tuple once an incomplete tuple t^* is computed by `GoodCore`, rather than conducting the imputation after K tuples are computed, as discussed in Section 4. To this end, we move the imputation step (lines 11-12 in Algorithm 1) inside the first loop of Algorthm 1, *i.e.*, imputing each selected t^* by a human or automatic method in each iteration after line 9.

Algorithm 3: Batch algorithm of `GoodCore`

Input: D, K, h, batch size b.
Output: A coreset C, weight $\mathbb{W}$.

```
1  C = ∅, cnt = 0;
2  while |C| < K do
3      Sample h tuples as T_sample ⊆ D \ C
4      for each tuple t ∈ T_sample do
5          E[t|C] = ComputeUtility(t, C, D);
6      t* = arg max_{t∈T_sample} E[t|C] ;
7      C = C ∪ {t*};
8      if 𝕀[t*] = 1 then
9          cnt++;
10     if cnt = b then
11         Ask the human to impute the incomplete tuples;
12         cnt = 0;
13 Compute the weight 𝕎.
14 return C, 𝕎;
```

Afterwards, we will add the next tuple into the coreset, so another loop starts and h tuples are sampled. In the following, we will expand the third loop, *i.e.*, the function `ComputeUtility` (line 7) of Algorithm 1 under this one tuple per iteration scenario.

As shown in Algorithm 2, at the beginning, we temporarily add the sampled tuple t to the current coreset, so as to compute the benefit of t, *i.e.*, $\mathrm{E}[t|C]$. To this end, we have to first compute the expectation of GA error bound of $\hat{C}$ (*i.e.*, computing $\mathrm{E}[\hat{C}]$ in the for-loop lines 3-9). And the expectation w.r.t. C (*i.e.*, $\mathrm{E}[C]$) has been computed in the last loop. Then we can compute $\mathrm{E}[t|C] = \mathrm{E}[C] - \mathrm{E}[\hat{C}]$ (line 10).

Specifically, to compute $\mathrm{E}[\hat{C}]$, we will use the tuple-based expectation computation method proposed in Section 4.2. For each tuple $t_i \in D$, if t_i and t are both complete, we can directly compute $\min_{c_j \in \hat{C}} s_{ij}$ because there is no incomplete data in $\hat{C}$ (lines 4-5). Otherwise, we will enumerate the possible worlds of $\hat{C} \cup \{t_i\}$, compute their probabilities and compute $\mathrm{E}[\min_{c_j \in \hat{C}} s_{ij}]$ (lines 7-8). Note that since there are at most two tuples (*i.e.*, t_i and t) have missing values, the number of possible worlds is small because other missing values in $\hat{C}$ have been imputed by humans in previous iterations.

Time complexity analysis. As discussed above, using this human-in-the-loop strategy, the number of possible worlds to be considered is greatly reduced. For Algorithm 2, the time complexity is $O(nL^2)$ because there are at most two incomplete tuples in $\hat{C}$. For the entire three loops framework, the time complexity is $O(KhnL^2)$, which is much lower than the solution without imputation in the loop.

However, if we utilize the human for imputation, the above method will incorporate many human iterations. In the following, we propose to ask human to impute a small batch of missing tuples in each iteration, so as to reduce the number of human iterations.

5.2 One Batch Each Iteration with Human-in-the-loop

In Section 5.1, one tuple per iteration by humans requires many human iterations. However, if we just incorporate a single human iteration like Section 4.2, it is infeasible to compute the tuples to be imputed due to the large number of possible worlds. Therefore, in this subsection, we propose a trade-off solution that asks the human to impute a small batch of tuples per human iteration.

To be specific, as shown in Algorithm 3, compared with the one tuple per human iteration algorithm (*i.e.*, the modified Algorithm 1 at the beginning of Section 5.1), we additionally take the batch size b as input (when $b = 1$, Algorithm 3 is in fact the modified Algorithm 1). Algorithm 3 also incorporates 3 loops, but the main difference is that we do not instantly ask the human to impute the most beneficial tuple t^* among T_{sample}. Instead, we just add t^* into the coreset C (line 7). When there have been b incomplete tuples, we ask the human to impute these tuples together (line 10-12). Finally we compute the weight (line 13), same as Algorithm 1. Although this approach reduces the number of human iterations, it takes a longer time to compute $\mathrm{E}[t|C]$ (line 5) than Algorithm 1 because there are more incomplete tuples, which indicates more possible worlds. Specifically, the time complexity of computing $\mathrm{E}[t|C]$ is $O(nL^b)$, which is also expensive. Hence, we propose a heuristic method to accelerate this process as follows.

Reducing the number of possible worlds. A straightforward method of improving the efficiency is to reduce the number of possible worlds. To this end, intuitively, we should focus more on the possible world with a high probability, so these possible worlds with low probabilities can be pruned without sacrificing the accuracy of expectation computation much. Note that for each possible world, the probability is computed by the multiplication of the probabilities of incomplete tuples in the world because the tuples can be considered independent [**?**]. Therefore, we can remove the possible worlds of each tuple with low probabilities (*i.e.*, reducing L), and thus the number of possible worlds of the entire coreset is greatly reduced. For example, we can keep top-l (*e.g.*, $l = 3$) possible worlds (*i.e.*, 3 different possible imputations of t with high probabilities) of a tuple t. Then for the batch of b incomplete tuples, the number of possible worlds is l^b and the complexity of computing $\mathrm{E}[t|C]$ is $O(nl^b)$, where both l and b are small enough. Overall, the time complexity is $O(Khnl^b)$. Besides, we can also apply this heuristic method to make the algorithm in Section 4 practical, which is evaluated in Section 6.4.

5.3 Convergence Rate Analysis

Convergence rate is often used to reflect the speed of finding the optimal parameters for the machine learning algorithm. With a higher convergence rate, we can take fewer epochs to make the model converge. To compute the convergence rate, we have to compute the distance between the parameter θ and the optimal parameter θ^* in the t-th and the $(t+1)$-th epoch. Since f is a strongly convex function, $\forall \theta, \theta'$ we have

$$f(\theta) - f(\theta') \geq \nabla f(\theta')(\theta - \theta') + \frac{\eta}{2}\|\theta - \theta'\|^2 \tag{8}$$

where η is a constant. We denote the stepsize as $\zeta_t = \frac{\zeta_0}{k^\tau}$ for the t-th epoch, where τ is a constant. After using gradient descent in each step, we have $\|\theta^{t+1} - \theta^*\|^2 = \|\theta^t - \zeta_k \sum_{j=1}^{|C|} w_j \nabla f_{\gamma(j)}(\theta^t_{j-1}) - \theta^*\|^2$. Then, following Eq. 8, we have

$$\begin{aligned} \|\theta^{t+1} - \theta^*\|^2 \leq \|\theta^t - \theta^*\|^2 - 2\zeta_t \sum_{j=1}^{|C|} (f_j(\theta^t) - f_j(\theta^*)) \\ +2\zeta_t \sum_{j=1}^{|C|} (f_j(\theta^t_{j-1}) - f_j(\theta^t)) + \zeta_t^2 \sum_{j=1}^{|C|} \|w_j \nabla f_j(\theta^t_{j-1})\|^2 \end{aligned} \tag{9}$$

Recap that we select a coreset that minimizes $\mathrm{E}[C]$ through converting gradient difference to feature distance (s_{ij}) computation. Obviously, given a dataset, s_{ij} can be bounded (suppose that $s_{ij} \leq s_0$). Then we have $\mathrm{E}[\min_{c_j \in C} s_{ij}] = \sum_{k=1}^{|\mathcal{I}_W|} p_k (\min_{c_j \in C_k} s_{ij}) \leq \sum_{k=1}^{|\mathcal{I}_W|} p_k * s_0 = s_0$, and thus $\mathrm{E}[C] = \sum_{i=1}^{n} \mathrm{E}[\min_{c_j \in C} s_{ij}] \leq n * s_0 = \kappa_1$. Besides, we also have $max_{\theta \in \vartheta} \| \sum_{i=1}^{n} \nabla f_i(\theta) -$

Table 1. Statistics of datasets

Dataset	$\|D\|$	M	# Incomp. Tuples	Task
Nursery	10960	9	3218	Multi-Class.
HR	18287	12	5475	Binary Class.
Adult	32842	14	10752	Binary Class.
Credit	131,000	11	76813	Binary Class.
BikeShare	13300	15	4821	Regression
Air	437,200	18	128,372	Regression

$\sum_{j=1}^{|C|} w_j \nabla f_{Y(j)}(\theta)\| \leq \sum_{i=1}^{n} \min_{c_j \in C} \|\nabla f_i(\theta) - \nabla f_{Y(j)}(\theta)\| \leq \sum_{i=1}^{n} \min_{c_j \in C} s_{ij} \leq \kappa_1$. Following the definition of convex function, we have $f_j(\theta^t) - f_j(\theta^*) \leq w_j \nabla f_j(\theta^*)(\theta^t - \theta^*) + \frac{\eta}{2}\|\theta^t - \theta^*\|^2$. Based on the above things, we can apply Cauchy-Schwarz inequlity [**?**] and derive

$$\begin{aligned} & -2\zeta_t \sum_{j=1}^{|C|}(f_j(\theta^t) - f_j(\theta^*)) \\ \leq & -\eta\zeta_t\|\theta^t - \theta^*\|^2 + 2\zeta_t\|\sum_{j=1}^{|C|} w_j \nabla f_j(\theta^*)\|\|(\theta^t - \theta^*)\| \\ \leq & -\eta\zeta^t\|\theta^t - \theta^*\|^2 + \frac{2\zeta^t|C|\kappa_1\kappa_2}{\eta} \end{aligned} \tag{10}$$

where κ_2 can be regarded as the upper bound of $\|\theta_t - \theta^*\|$. Since f is convex, thus, for item $f_j(\theta^t_{j-1}) - f_j(\theta^t)$, we have $f_j(\theta^t_{j-1}) - f_j(\theta^t) \leq \|w_j \nabla f_j(\theta^t)\|\zeta_t \sum_{i=1}^{j-1} \|w_i \nabla f_i(\theta^t_{i-1})\|$. In addition, we can assume that $max_{j\in\{1,\cdots,|C|\}}\|\nabla f_j(\theta)\| \leq \kappa_3$. Then, we have

$$\begin{aligned} & 2\zeta_t \sum_{j=1}^{|C|}(f_j(\theta^t_{j-1}) - f_j(\theta^t)) + \zeta_t^2 \sum_{j=1}^{|C|}\|w_j \nabla f_j(\theta^t_{j-1})\|^2 \\ \leq & 2\zeta_t \sum_{j=1}^{|C|}\|w_j \nabla f_j(\theta^t)\|\zeta_t \sum_{i=1}^{j-1}\|w_i \nabla f_i(\theta^t_{i-1})\| + \zeta_t^2 \sum_{j=1}^{|C|}\|w_j \nabla f_j(\theta^t_{j-1})\|^2 \\ \leq & 2\zeta_t^2(|C|^2 - |C|)w_{max}^2\kappa_3^2 + \zeta_t^2|C|w_{max}^2\kappa_3^2 \end{aligned} \tag{11}$$

Thus, from Eq. 9 to Eq. 11, we can get

$$\|\theta^{t+1} - \theta^*\| \leq (1 - \eta\zeta_t)\|\theta^t - \theta^*\|^2 + \frac{2\zeta_t|C|\kappa_1\kappa_2}{\eta} + \zeta_t^2|C|^2 w_{max}^2\kappa_3^2 \tag{12}$$

Finally, following Lemma 4 in [**?**], the convergence rate of Algorithm 1 is at the same rate of $O(\frac{1}{\sqrt{k}})$ as the convergence rate on the entire dataset [**?**]. Therefore, theoretically, the selected coreset can converge with the same number of epochs as training on the full data. In this way, since coreset has a much smaller size than the full data, the efficiency can be much improved.

6 EXPERIMENT

In this section, we sufficiently compare our proposed methods with multiple baselines on real datasets to demonstrate our effectiveness and efficiency.

6.1 Experimental Settings

Dataset. We evaluate on 6 real-world datasets that are often used in the field of data imputation [**? ? ? ?**], as shown in Table 1, where M denotes the number of attributes.

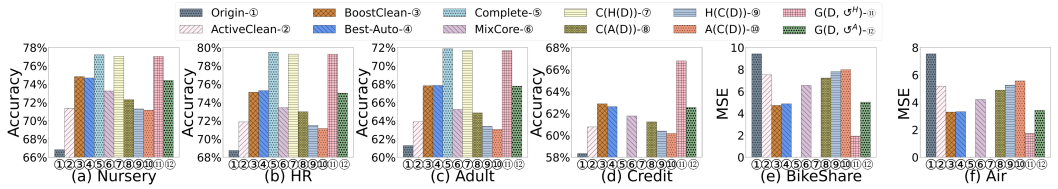

Fig. 8. Effectiveness of different methods.

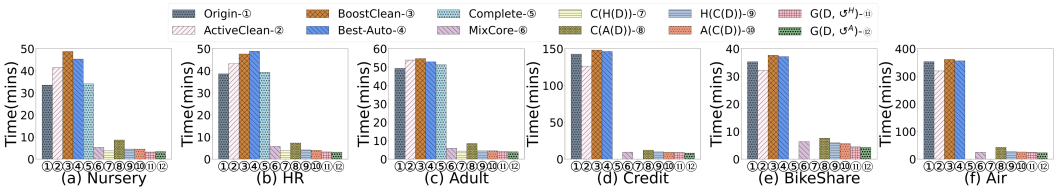

Fig. 9. Efficiency of different methods. Note that only machine cost (i.e., runtime of machine) is considered.

(1) `Nursery` [**?**] is a multi-classification task, which predicts "*the level of recommendation for whether a child goes to school*". There are five different levels, *i.e.*, {`not_recom`, `priority`, `recommend`, `spec_prior`, `very_recom`}. **(2)** `HR` [**?**] is a binary classification task of "*predicting whether an employee would change the job*". **(3)** `Adult` [**?**] is a binary classification task that predicts "*if the annual revenue of a people is over 50000 dollars*". **(4)** `Credit` [**?**] is a binary classification task that predicts "*whether the loan will be deferred based on a person's economic situation*". **(5)** `BikeShare` [**?**] is a regression task that predicts "*the number of bike sharing in a given time*". **(6)** `Air` [**?**] is a regression task that predicts "*the air quality at a certain time*".

For datasets (1)-(3), we follow existing works [**? ? ?**] to manually inject missing values until the rate of missing tuples is 30%, and we will vary the percentage of incomplete tuples in Section 6.6. Datasets (4)-(6) already contain missing values. For all datasets, we randomly split them for 80%/10%/10% as train/validation/test sets.

Evaluation metrics. We mainly evaluate the effectiveness and efficiency of `GoodCore` and baselines. For effectiveness, we use the *prediction accuracy* for the classification task and use the *mean square error* ($MSE = \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)}{N}$, where N denotes the size of test set) for the regression task.

For efficiency, we focus on the machine cost (*i.e.*, the runtime of machine) as well as the human cost (the number of tuples imputed by human for human-involved methods). For datasets (1)-(3), we have the ground truth of missing tuples, so we use them to simulate the human imputation. For datasets (4)-(6), we leverage the expert to impute missing values in the coreset by looking at the top-5 values recommended by the automatic method as a reference. Note that we only involve humans when it is affordable. For baselines that require humans to impute a lot of missing tuples (*i.e.*, `Complete` and $\mathbf{C}(\mathbf{H}(D))$ as below), we will not apply them on datasets (4)-(6).

Baselines. We compare `GoodCore` with a variety of baselines.

(1) `Origin` refers to just training on D.

(2) `ActiveClean` [**?**] is an iterative data cleaning framework, which estimates the impact of tuples and prioritizes cleaning the tuples that much affect the model performance. In each iteration, it can ask the human to clean a sample subset of tuples. We set the sample size to 50, same as the paper.

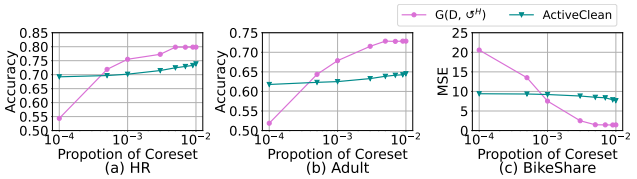

Fig. 10. Coreset size selection of GoodCore.

(3) BoostClean [**?**] is an automatic data cleaning method that iteratively selects a cleaning method from several pre-defined algorithms, applies to the train dataset and updates the model. We use MICE [**?**], MISSForest [**?**], GAIN [**?**] as pre-defined algorithms.
(4) Best-Auto uses MICE [**?**], MISSForest [**?**], GAIN [**?**] to respectively impute the train set and selects the one that achieves the highest accuracy on the validation set.
(5) Complete is an ideal case that trains on the ground truth, *i.e.*, D_c. Note that only datasets (1)-(3) have the ground truth to evaluate this baseline. Datasets (4)-(6) do not have the ground truth and it is too expensive to ask the human to impute so many missing values.
(6) MixCore is a baseline that selects a coreset from all complete tuples, and then we randomly select some incomplete tuples to impute. We set the number of incomplete tuples to be imputed equal to that of other baselines for fair comparison. Finally we train with the tuples in the coreset plus the imputed ones.
(7) $\mathbf{C}(\mathbf{H}(D))$ first involves human to impute the dataset D and then selects a coreset. Similar to Complete, only datasets (1)-(3) can be evaluated on it because they have the ground truth. The coreset selection solution is the algorithm in [**?**], which is a greedy algorithm by modifying Algorithm 1 without considering the possible worlds.
(8) $\mathbf{C}(\mathbf{A}(D))$ first uses automatic data imputation methods to impute the dataset D, and then selects a coreset using the same method of baseline (7).
(9) $\mathbf{H}(\mathbf{C}(D))$ directly selects a coreset based on D and then asks human to impute the incomplete tuples of the coreset.
(10) $\mathbf{A}(\mathbf{C}(D))$ also directly selects a coreset from D, it then uses MICE [**?**] to impute the incomplete tuples in the coreset.
Our solutions. We compare GoodCore and its variants.
(11) $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ uses GoodCore to select the coreset and iteratively asks human to impute incomplete tuples (one tuple per human iteration) during the coreset selection process.
(12) $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ is similar to $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$, but the automatic MICE method is used.

Besides, since the coreset of $\mathbf{H}(\mathbf{G}(D))$ (or $\mathbf{A}(\mathbf{G}(D))$) is too expensive to compute due to the large number of possible worlds, we do not directly compare with it. Instead, we will limit the number of possible worlds of each tuple to 3 as discussed in Section 5.2 and evaluate in Section 6.4.
Hyper-parameter setting. We use SVM and linear regression as the default downstream model for classification and regression tasks, respectively. We vary the downstream models in Section 6.6. For model training, we use SGD and k-inverse decay scheduling, *i.e.*, $\alpha_k = \alpha_0/(1 + bk)$ (α_0 and b are hyper-parameters to be tuned independently for different methods). The sample size h is set to 200 as default and we vary the size in Section 6.6. The number of training epochs is set as 20. We also impute the test data using the same method that is applied to the train data before testing.

6.2 Overall Evaluation

In this part, we compare GoodCore solutions with baselines. We use $\rho = \frac{K}{|D|}$ to denote the proportion of coreset to the entire train set. We set $\rho = 0.005$ for datasets (1)-(4), $\rho = 0.001$ and $\rho = 0.0005$

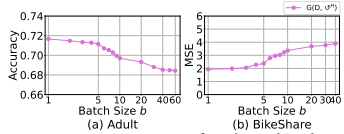

Fig. 11. GoodCore for batch algorithm.

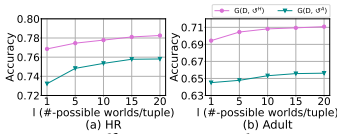

Fig. 12. Effectiveness when varying l.

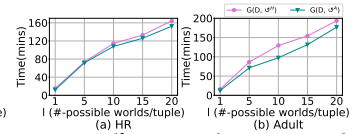

Fig. 13. Efficiency when varying l.

Table 2. Human cost of different methods

Dataset	$\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$	$\mathbf{H}(\mathbf{C}(D))$	$\mathbf{C}(\mathbf{H}(D))$
Nursery	37	22	3278
HR	44	32	5475
Adult	63	81	10752
Credit	52	67	-
BikeShare	38	25	-
Air	98	102	-

for larger datasets (5) and (6) respectively. We will further conduct evaluation by varying the coreset size in Section 6.3.

6.2.1 Evaluation of model accuracy. The results are provided in Figure 8. To summarize, the result could be generally ranked as $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})/\mathbf{C}(\mathbf{H}(D))/$Complete $>$ $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})/$BoostClean/Best $-$ Auto $>$ $\mathbf{C}(\mathbf{A}(D))$ $>$ MixCore $>$ ActiveClean $>$ $\mathbf{H}(\mathbf{C}(D))/\mathbf{A}(\mathbf{C}(D))$ $>$ Origin. Next, we explain the results.

In general, on all datasets, our method $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$, Complete and $\mathbf{C}(\mathbf{H}(D))$ perform the best. Complete and $\mathbf{C}(\mathbf{H}(D))$ achieve a high accuracy because they ask the human to impute missing values accurately, but incur a high human cost. For example, Complete and $\mathbf{C}(\mathbf{H}(D))$ achieve accuracy of 71.9% and 71.7% on Adult. $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ is competitive with them because it selects a good coreset that can well represent the unknown ground truth via gradient approximation. In addition, we can observe that $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ performs better than $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ because human imputation is more accurate than automatic methods. For example, on Adult, $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ has an accuracy of 71.7%, while $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ and others are below 68%. $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$, BoostClean and Best-Auto have competitive performance on accuracy. BoostClean and Best-Auto can have a not bad performance because they impute all tuples and train on the entire dataset, but they cannot achieve efficient training (see 6.2.2). But we can train on the much smaller coreset generated by $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ with a good accuracy, because GoodCore considers the possible repairs to derive the coreset that can approximate the full gradient of the entire dataset. Given the same number of tuples to be imputed by human, $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ also outperforms ActiveClean because we have theoretical guarantees on the gradient approximation. For other baselines, $\mathbf{H}(\mathbf{C}(D))$ and $\mathbf{A}(\mathbf{C}(D))$ do not perform well because they select the coreset from an incomplete dataset. $\mathbf{C}(\mathbf{A}(D))$ cannot achieve a good performance because the selected coreset can not well represent the complete entire dataset, as it does not consider possible repairs as our method. MixCore does not perform well (*e.g.*, 65.2% on Adult) because $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ and $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ select a better coreset considering the full data. For Origin, on Adult, the model has an accuracy of 61.3% because of the incomplete tuples.

6.2.2 Evaluation of the efficiency. We evaluate the efficiency of all methods, including the machine cost and human cost.

Machine cost. Machine cost is shown in Figure 9. The results could be ranked as $\mathbf{H}(\mathbf{C}(D))/\mathbf{A}(\mathbf{C}(D))/\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})/\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})/\mathbf{C}(\mathbf{H}(D))/$MixCore $<$ $\mathbf{C}(\mathbf{A}(D))$ $<$ Complete $<$ Origin $<$ ActiveClean $<$ BoostClean/Best $-$ Auto. We can observe that the first 5 methods in the ranking have low machine cost, mainly because they train based on the selected coreset

and do not need iterative training. $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ and $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ are slightly slower because they need to iterate several possible repairs during the process of coreset selection. But $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ is still more efficient than `Origin`, `Complete`, `BoostClean` and `Best-Auto` by more than one order of magnitude, because they need to train on the entire training data. Moreover, `ActiveClean` and `BoostClean` are not efficient either because they incorporate multiple training times, so as to estimate the gradient while data imputation. `Best-Auto` is slow because training multiple imputation models takes time.

Human cost. In terms of the human cost, $\mathbf{C}(\mathbf{H}(D))$, $\mathbf{H}(\mathbf{C}(D))$, $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ and `ActiveClean` involve human. As shown in Table 2, $\mathbf{C}(\mathbf{H}(D))$ is very expensive because it asks the human to impute all missing tuples. For example, on datset `Adult`, 10752 tuples have to be imputed. We do not compare `Credit` , `BikeShare` and `Air` for $\mathbf{C}(\mathbf{H}(D))$ because they do not have the ground truth. But $\mathbf{H}(\mathbf{C}(D))$ and $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ are cost-effective because human just needs to impute missing tuples in the much smaller coreset. For example, they only cost 81 and 63 tuples on dataset `Adult` respectively. `ActiveClean` asks the human to iteratively impute the data. Given the same number of tuples to impute, our method can achieve much higher accuracy. We will evaluate it in details in next subsection.

Summary. Based on the results, we have the following conclusions. (1) Our proposed methods $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ and $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ can achieve high model accuracy because the selected coreset can well represent the underlying ground truth by gradient approximation considering possible repairs. Meanwhile, they are practical because of the low machine cost. (2) Compared with $\mathbf{C}(\mathbf{H}(D))$ that involves human to impute the entire dataset D, the human cost of $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ is much lower, as observed in Table 2, e.g., 37 vs. 3278 on the `Nursery` dataset. Thus, we can choose $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ when we want to achieve high model accuracy and afford a certain human cost. (3) If we neither care very much about the accuracy nor consider to incur human cost, the much more efficient $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ is a good choice.

6.3 Coreset Size Selection of `GoodCore`

Recap that `GoodCore` needs the user-specified coreset size as input. Thus, we discuss how to select an appropriate coreset size. We adopt a simple yet effective solution that starts from a coreset with a small size, train over it and evaluate via a validation set, enlarge the coreset and iteratively train until the performance does not improve much. To be specific, initially, we begin with $\rho = 10^{-4}$, and enlarge the coreset by 2 times iteratively. If the performance on validation set varies no more than 0.5% within three successive iterations, we will stop. Figure 10 shows the performance on dataset `HR` , `Adult` and `BikeShare` when varying the coreset size. We can see that the performance first improves rapidly, then remains stable just after several iterations. For example, on dataset `Adult`, when $\rho = 5 \times 10^{-3}$, the accuracy has improved to 72.85% on the validation set. Empirically, an ideal coreset size is between $\rho = 10^{-3}$ to 10^{-2}.

Summary. The results show that coreset size is not difficult to determine. If the user is willing to specify a coreset size like in Section 6.2 based on the empirical finding, we can directly compute a coreset without training. If she cannot, we can also get a good coreset with just several training iterations over small coresets, which is also efficient.

Compare with `ActiveClean`. Figure 10 also reports an interesting comparison with `ActiveClean`. Specifically, in `ActiveClean`, we use the coreset size K as the budget, i.e., number of tuples to be imputed by human in each active cleaning iteration. We can observe that at the beginning, when the coreset size is very small, `ActiveClean` is better because it trains with the entire dataset including the imputed tuples, while we train the model using only few tuples in the coreset. However, as with the increase of the coreset size, we can see that $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ outperforms `ActiveClean`. This is because `ActiveClean` uses a heuristic method to estimate the impact of tuples to the overall

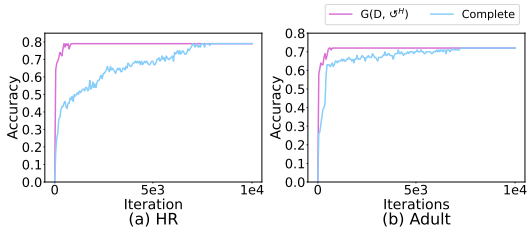

Fig. 14. Convergence of GoodCore.

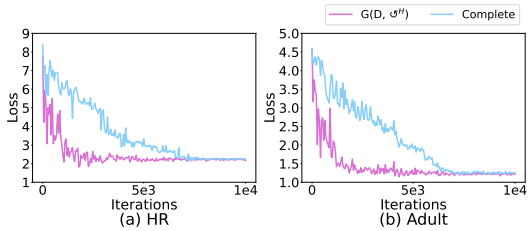

Fig. 15. Loss of GoodCore.

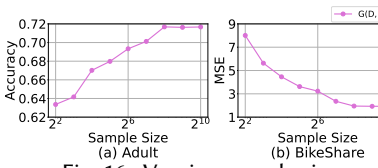

Fig. 16. Varying sample size.

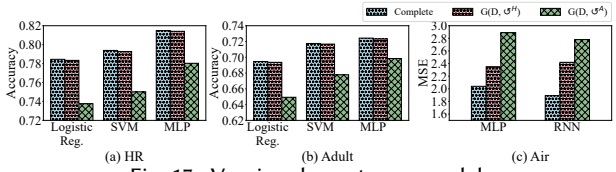

Fig. 17. Varying downstream model.

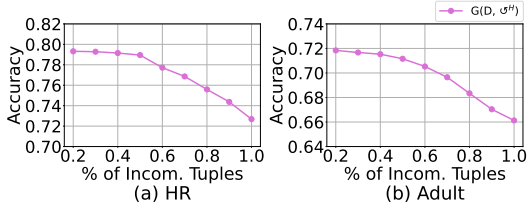

Fig. 18. Varying missing tuple rate.

gradient, which is not theoretically bounded (e.g., with gradient bounds like Coreset) and thus not accurate enough. For $\mathbf{G}(D, \mho^{\mathbf{H}})$, it can achieve high accuracy with a proper coreset size, which is not large.

6.4 Batch Algorithm of GoodCore

In Section 6.2, $\mathbf{G}(D, \mho^{\mathbf{H}})$ outperforms other baselines on accuracy, but requires many human iterations. In this part, we evaluate the batch algorithm of GoodCore by varying the batch size b, *i.e.*, Algorithm 3 to reduce the number of iterations. Intuitively, the algorithm is $\mathbf{G}(D, \mho^{\mathbf{H}})$ when $b = 1$. Then we increase b until a single batch with a size b can contain all incomplete tuples in the coreset with size K, which is in fact the algorithm $\mathbf{H}(\mathbf{G}(D))$. Due to the large number of possible worlds, we adopt the heuristic method in Section 5.2 to set $l = 3$ when $b > 1$.

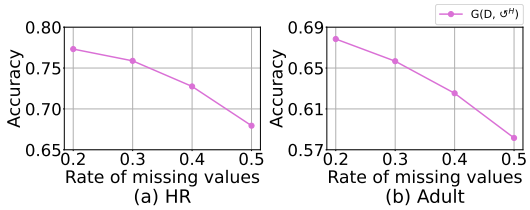

Fig. 19. Varying missing value rate.

Table 3. The number of possible worlds on different datasets

Method	Nursery	HR	Adult
$\mathbf{H}(\mathbf{G}(D))$	10^{201}	10^{201}	10^{202}
$\mathbf{A}(\mathbf{G}(D))$	10^{201}	10^{201}	10^{202}
$\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$	10^3	10^3	10^4
$\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$	10^3	10^3	10^4

In Figure 11, the x-axis denotes the batch size and the y-axis denotes the test performance on dataset `Adult` and `BikeShare` . We can see that when b is small (*i.e.*, $b \leq 5$), the performance does not significantly decrease (*e.g.*, on `Adult`, the accuracy decrease from 71.7% to 71.2%.). When b keeps increasing, the performance slightly decreases. Thus, `GoodCore` is not sensitive to the batch size b and the Algorithm 3 can reduce the number of human iterations without sacrificing much model performance.

In this part, we also vary the number of possible worlds by varying l, which is the number of possibles world per tuple. The larger l, the larger number of possible worlds we have. The results are shown in Figures 12 and 13. In terms of the accuracy, we can see that with l increasing (fixing $b = 10$), the accuracy increases first and then remains stable soon, but the time keeps increasing because more possible worlds indicate more computation. Hence, we do not need a large l.

When it comes to the number of possible worlds, we would like to clarify that we do not compare with $\mathbf{H}(\mathbf{G}(D))$ and $\mathbf{A}(\mathbf{G}(D))$ because the number of possible worlds of D is very large, which is infeasible to compute. We show the number in Table 3, where we also report the numbers of possible worlds of $\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$ and $\mathbf{G}(D, \circlearrowleft^{\mathbf{A}})$ in each iteration, which are practical to compute.

6.5 Convergence Evaluation

In Section 5.3, we have shown the convergence rate of `GoodCore` theoretically. In this part, we test the convergence of training over the coreset ($\mathbf{G}(D, \circlearrowleft^{\mathbf{H}})$) and entire data (`Complete`) empirically. Figure 14 shows the test accuracy of two methods with the number of training iterations increasing. We can observe that on both datasets, training on the coreset converges much faster than training on the full data. For example, on dataset `Adult`, it takes ~40 iterations for `GoodCore` to converge, which is 180× faster than `Complete`. This is because `GoodCore` has the same convergence rate with training over the entire dataset as discussed in the theoretical result of Section 5.3, but the entire dataset (*e.g.*, `Adult`) is 200× (similar to 180×) larger than the coreset ($\rho = 0.005$). That is, `GoodCore` converges with the same number of epochs as training on the entire dataset. Since the size of coreset is much smaller, `GoodCore` is more efficient. Also, we can achieve competitive accuracy as training on full data by approximating the full gradient with a theoretical bound.

Furthermore, we report the loss change to reflect the relation between actual convergence rate and theoretical results. In Figure 15, on dataset `Adult`, the initial loss is 8.4. According to the theoretical convergence rate $O(\frac{1}{\sqrt{k}})$ (this k denotes the k-th epoch), the loss should decrease to

around 3.8 at the end of 5-th epoch ($\approx$ 3200-th iteration). Actually, the actual loss decreases to 3.25 at that time, which is close to the theoretical value.

6.6 Sensitivity Analysis

Varying the sample size. In this part, we vary the sample size h and evaluate the performance. The experimental results are shown in Figure 16. We vary the sample size h from 2^2 to 2^{10}. We can see that when h is too small, the performance is low. The reason is that GoodCore cannot precisely estimate the utilities of tuples when h is small. When the sample size increases, we can see that the performance improves rapidly and finally becomes stable, which indicates that GoodCore is not much sensitive to the sample size when h is not too small.

Varying the downstream models. Recap that GoodCore can be used on different convex models. Thus, in this part, we apply GoodCore on different convex models and evaluate the performance. We evaluate logistic regression and SVM for classification tasks. For regression tasks, we evaluate linear regression, ridge regression and SVM regression. We can see that in Figure 17 (a) and (b), on dataset Adult, $\mathbf{G}(D, \cup^{\mathbf{H}})$ achieved 71.7% accuracy for SVM, better than on logistic regression (69.4%). Although different downstream models may have different performance, GoodCore can improve the model performance for the specific downstream model. In order to show that GoodCore can be used for other types of models like neural networks, we compare with Multilayer Perceptron (MLP, fully connected networks of 2 hidden layers with 256 nodes for each layer), although GoodCore does not hold theoretical guarantee for this non-convex model. As shown in Figure 17, we can see that MLP achieves almost the same performance as the ground truth. This is because the coreset selected by GoodCore can also represent the full dataset. However, in Figure 17(c), on a large dataset Air (with metric MSE, the lower the better), neural network based methods (we also implement RNN, 2 hidden layers with 128 nodes for each layer) can have a better accuracy but the coreset cannot perfectly achieve the same performance. This may because this large dataset has more informative things to learn, and it is hard for the coreset-based solution to well represent the dataset without the theoretical guarantee.

Varying the percentage of incomplete tuples. In this part, we vary the rate of missing tuples and evaluate the performance, as shown in Figure 18. Note that the rate denotes the percentage of incomplete tuples rather than the cell values. Even if a tuple just has one missing attribute, it is regarded as incomplete. We vary the percentage from 20% to 100%. We can observe that the performance does not decrease much with the percentage increasing from 20% to 50%, which indicates that GoodCore is not very sensitive to the percentage of incomplete tuples in this range. After that, the accuracy decreases because there are more missing tuples.

Besides, we also vary the rate of missing cell values in Figure 19. In this scenario, for example, 50% missing values of a dataset indicates more number of missing cell values than the scenario of 50% missing tuples. Hence, we can see that the accuracy decreases more quickly than Figure 18.

7 RELATED WORK

Task-agnostic incomplete data imputation. Data imputation has been widely studied for years. Existing methods can be divided into two categories: statistic-based methods and learning-based methods. The former one always uses the statistic information [**?** **?**] (like mean, median or mode) to impute the missing values. Also, some methods compute the similarity of the incomplete tuples to the complete tuples and use the most similar one to impute the missing values [**?** **?** **?**]. Recently, to improve the imputation accuracy, many learning-based methods focus on how to use ML to learn the data distribution (*e.g.*, MissForest imputation [**?**], MICE [**?**], IIM [**?**]), and then use the trained model to predict the missing values. Besides traditional ML models, some deep learning models are also used for data imputation (*e.g.*, autoencoder [**?** **?** **?**], GANs [**?** **?**]).

Coreset selection. Huang et al. [**?**] studied how to compute and continuously update the coreset while training. But it is rather time-consuming because of the training process. To solve this problem, works [**? ?**] selected the coreset without training in advance, but they can only be customized to particular model types respectively. Another line of works [**? ? ?**] focused on selecting the coreset to approximate the full gradient without training in advance for multiple model types, which is regarded as an optimization problem that can be solved by the three nested loops framework (see Section 2.2).

In short, none of them considers coreset selection over incomplete data. Different from them, we make the first attempt to select coresets over incomplete data.

Data cleaning for ML. Recently, there have been several works that clean the data to optimize the ML model. In contrast to the above discussion about task-agnostic incomplete data imputation, data cleaning for ML is task-aware, which triggers new technical challenges. SampleClean [**?**] focuses on cleaning selected samples, so as to answer SQL aggregate queries more efficiently, but it is not for any model. CPClean [**?**] proposes certain prediction to impute missing data for optimizing ML models. Different from us, it is customized to nearest neighbor classifiers rather than convex models solved by the gradient decent algorithm. BoostClean [**?**] regards data cleaning as a boosting problem that iteratively selects from a predefined set of cleaning algorithms, so as to continuously maximize the accuracy of a validation set with training iteratively. Closer to our work is ActiveClean [**?**], which progressively cleans the data tuples that are likely to much influence the model measured by the gradients. Different from us, given a budget K, we can select the coreset without training, but ActiveClean needs to train iteratively and label a set of validation dataset. We empirically show that our method outperforms ActiveClean on model accuracy and efficiency in Section 6.

Data Preparation for AI. Data preparation [**?**] can be utilized to improve the effectiveness of ML model, including data discovery [**? ? ?**], data cleaning [**? ?**], data labeling [**? ? ?**], data cleaning [**? ? ? ? ? ?**] and data exploration [**? ?**].

8 CONCLUSION

In this paper, we propose the `GoodCore` framework to select a good coreset over the incomplete data, which achieves data-effective and data-efficient ML. We formulate it as an expected optimal coreset selection problem, which is NP-hard. Then we propose a greedy algorithm with an approximation ratio. We also propose to involve imputation-in-the-loop strategies into `GoodCore` to further improve the efficiency. We conduct experiments on real-world datasets to verify the effectiveness and efficiency of `GoodCore`.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009