CLEANCORE: DATA-QUALITY-AWARE CORESET SELECTION FOR LEARNING OVER DIRTY TABULAR DATA

Anonymous authors
Paper under double-blind review

ABSTRACT

Multiple error types and high training costs in large-scale tabular data make it difficult to balance predictive performance and runtime efficiency. Existing data cleaning, coreset selection, and hybrid methods typically focus on one objective or specific error types. To address this, we propose **CleanCore**, a data-quality-aware coreset selection method that handles multiple types of errors. The method iteratively uses training signals for error typing and lightweight handling, and dynamically maintains training coresets based on sample reliability. Theoretical analysis provides support from three perspectives: error-typing reliability, error-handling bias, and coreset representativeness. Comparisons with eleven baselines on eleven public tabular datasets using MLP and tabular ResNet show that **CleanCore** achieves the best ACC and F1 in 21 and 20 of the 22 model–dataset settings, respectively, delivers approximately 2–24$\times$ runtime speedups over three data cleaning methods, and maintains strong performance under real-world dirty-data conditions and different injected-error settings. Further experiments validate its stability and the effectiveness and efficiency of its main mechanisms.

1 INTRODUCTION

Modern deep learning typically relies on large-scale data to learn effective representations and improve predictive performance (Halevy et al., 2009; Zha et al., 2025; LeCun et al., 2015). However, real-world data often contain errors (Rahm et al., 2000; Emmanuel et al., 2021; Song et al., 2022). According to the affected object, these errors can be divided into label errors, feature errors, and mixed errors; according to detectability, they can be divided into explicit and implicit errors. Meanwhile, increasing data scale substantially raises computation and model training costs (Schwartz et al., 2020; Strubell et al., 2019), making full-data processing and training more difficult. Large-scale training thus faces the dual pressures of inadequate data quality and high computational costs. Selecting a small, high-quality, representative training coreset from the original data could balance predictive performance and training efficiency.

Existing related approaches fall into three main categories: data cleaning methods, coreset selection methods, and hybrid methods, which focus on data quality, training efficiency, and their combination, respectively. Further discussion of related work is provided in Appendix A. Although coreset selection reduces training data size, the selection process itself may incur substantial overhead. We therefore focus on overall runtime efficiency across both data processing and model training. How to balance model performance and overall runtime efficiency on large-scale data containing multiple error types remains insufficiently resolved. The following example illustrates this problem.

Example 1. Credit card transaction fraud detection. Figure 1 illustrates this task, where x_1 denotes the transaction amount and x_2 denotes the transaction time. Detecting fraudulent transactions among millions of records collected from multiple financial sources is a typical large-scale classification task. The three categories perform differently in predictive performance and overall runtime efficiency. Cleaning methods require additional full-data processing and subsequent training, incurring substantial overhead (c_2). Coreset selection may select erroneous records as representatives, affecting training (c_3). Hybrid methods such as GoodCore (Chai et al., 2023) and Deem (Deng et al., 2025) typically handle only specific error types and cannot readily cover cases where multiple error types coexist (c_4).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

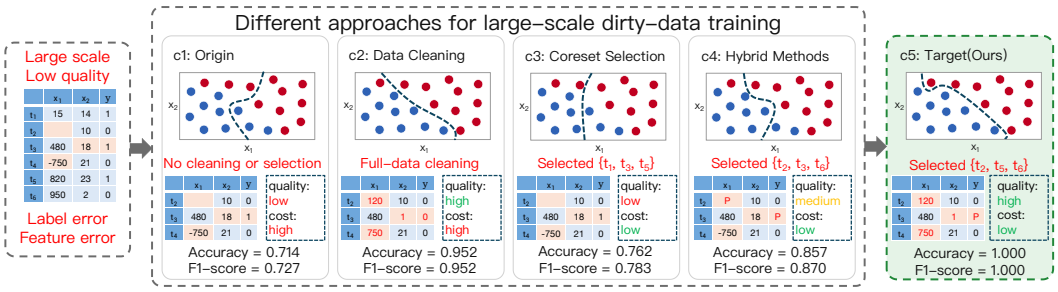


Figure 1: Model performance and overall runtime efficiency in credit card fraud detection. P denotes a placeholder for a processed value. Selected lists coreset sample IDs; the small tables illustrate the handling of fixed records, not coreset membership.

A simple extension combines data cleaning and coreset selection sequentially. However, cleaning before selection requires additional full-data processing, while selecting before cleaning allows original data errors to affect coreset selection. Another approach extends GoodCore or Deem by jointly considering candidate labels and candidate feature repairs, but this incurs high computation and memory costs that may offset the efficiency gains of coreset training. Simply chaining or extending existing approaches therefore remains insufficient. A coreset selection method is needed that addresses multiple error types while improving representativeness and overall runtime efficiency.

To meet this need, we observe that training signals can be used to identify different error types and apply targeted handling. A coreset can then be dynamically selected according to the reliability and representativeness of the processed samples and used for subsequent training, improving overall runtime efficiency with little additional overhead. **This forms an iterative feedback process** that achieves the target state in c_5 of Figure 1.

However, implementing this idea reliably and efficiently still presents the following challenges. **C1: Reliable error typing is difficult when training signals are unstable. C2: Handling multiple error types must balance efficiency and effectiveness. C3: Coreset selection is difficult to make reliable under noisy signals.**

We propose CleanCore, a data-quality-aware coreset selection method for training on large-scale tabular data containing multiple error types. Our main contributions are as follows:

(1) We formalize **data-quality-aware coreset selection for dirty tabular data** and propose a staged iterative framework that uses training signals to continuously update sample reliability weights and the coreset, reducing training costs while maintaining reliability and representativeness.

(2) We design **ErrorTyping**, a **training-signal-driven error typing mechanism** that combines recent loss statistics, label replacement, and feature-gradient signals to distinguish multiple error types while seeking to reduce misidentification of hard but correct samples.

(3) We propose **ErrorHandle**, an **error-type-guided lightweight error handling strategy** that applies differentiated treatments to identified errors and dynamically adjusts the reliability weights of processed samples to efficiently suppress the effects of errors.

(4) We propose **CoreSelect**, a **reliability-weighted dynamic coreset selection method** that uses reliability-weighted gradient coverage and adaptive updating to maintain coreset representativeness and improve training efficiency under noisy signals.

(5) We compare CleanCore with eleven baselines on eleven public tabular datasets using MLP and tabular ResNet. CleanCore achieves the best ACC and F1 on the vast majority of datasets under both models and provides about 2–24$\times$ runtime speedups over three data cleaning methods. Further experiments validate its stability and its main mechanisms' effectiveness and efficiency.

CleanCore targets deep learning classifiers for tabular data optimized through gradients, with explicit errors preprocessed and then handled together with implicit errors. Supplementary explanations of the research background, example and methodological motivation, challenges and contributions, scope of applicability, and error preprocessing in this section are provided in Appendix B.

2 OVERVIEW

This section introduces the basic concepts, problem definition, and overall framework. Formal definitions and supplementary explanations of the basic concepts, together with the problem analysis, are provided in Appendix C.1; supplementary explanations of the method framework are provided in Appendix C.2.

2.1 PRELIMINARIES AND PROBLEM DEFINITION

Preliminaries. We study supervised learning on structured tabular data (Borisov et al., 2022). For a dataset $D = \{(x_i, y_i)\}_{i=1}^{n}$ of n samples, the model takes a d-dimensional feature vector $x_i \in \mathbb{R}^d$ as input and predicts its label y_i from the class set Y, where $|Y|$ is the number of classes. The differentiable predictor f_θ learns parameters θ by minimizing the training loss (Bottou et al., 2018), with individual-sample loss $\ell_i(\theta) = \ell(f_\theta(x_i), y_i)$. In practice, a sample's features, label, or both may deviate from its true clean form (x_i^*, y_i^*), resulting in feature, label, or mixed errors that affect model training. Data cleaning aims to reduce these effects by transforming the original dataset D into a processed dataset D'. Meanwhile, full-data training is costly at scale. Coreset selection selects a representative subset C satisfying $|C| \leq K$ under budget K and assigns sample weights $\widetilde{w}$, using the weighted coreset $(C, \widetilde{w})$ to approximate full-data training behavior and reduce computation.

Problem Definition. For large-scale dirty-data training, let D denote the currently available training pool and $K \ll n$. The current model parameters θ take the initial value $\theta^{(0)}$ before training and the current parameter state during training.

Definition 1 (Data-quality-aware coreset selection problem for dirty tabular data)**.** *Given a dirty training pool D, current model parameters θ, and budget K, the goal is to select a reliable, representative coreset C from D and determine its training weights $\widetilde{w}$, namely $(C, \widetilde{w}) = \mathrm{Select}(D, \theta, K), |C| \leq K$. Here,* Select *denotes data-quality-aware coreset selection, and $\widetilde{w}$ regulates the selected samples' contributions to model updates.*

The coreset and its weights may be adjusted as the model state and sample reliability change and are used for subsequent training, yielding parameters $\widehat{\theta}_C = \mathrm{Train}(\theta, C, \widetilde{w})$. Let D^* be the ideal high-quality counterpart of D, θ^{full} the reference parameters obtained by training on D^*, Perf the chosen task metric, and $\varepsilon_{\mathrm{perf}}$ the allowed performance tolerance. The goal is to satisfy $\mathrm{Perf}(\widehat{\theta}_C) \geq \mathrm{Perf}(\theta^{\mathrm{full}}) - \varepsilon_{\mathrm{perf}}$ while achieving $\mathrm{TIME}_{\mathrm{core}} \ll \mathrm{TIME}_{\mathrm{full}}$, where the latter two quantities denote the total runtimes with the coreset and full dataset, respectively.

2.2 METHOD FRAMEWORK

The preceding problem definition and the analysis in Appendix C.1 indicate that data-quality-aware coreset selection should iteratively construct and update the coreset as the model state and sample reliability change. Following this idea, CleanCore adopts a staged iterative framework, as shown in Figure 2. CleanCore first trains on the full dataset for several epochs to obtain preliminary losses, gradients, and model states. It then executes the following three components in sequence at each stage t; the processed sample pool and the coreset constructed or updated at this stage use the superscript $t + 1$.

Training-signal-driven error typing (ErrorTyping). Using recent model snapshots, it screens persistent high-risk samples and identifies error types through label-consistency and feature-sensitivity checks, temporarily assigning samples whose types remain unclear to the mixed-error candidate set.

Error-type-guided lightweight error handling (ErrorHandle). It applies differentiated treatments according to error types and updates reliability weights, removing only current high-risk samples whose updated weights fall below the removal threshold.

Reliability-weighted dynamic coreset selection (CoreSelect). Using the gradient representations and reliability weights of processed samples, it checks representativeness at stage boundaries and obtains a coreset through reliability-weighted gradient coverage or adaptive updating.

Let the processed sample pool contain $n^{(t+1)}$ samples, with its i-th sample denoted by $z_i^{(t+1)}$. The coreset at this stage is $C^{(t+1)}$, with training weights $\widetilde{w}^{(t+1)} = \{\widetilde{w}_j^{(t+1)} : z_j^{(t+1)} \in C^{(t+1)}\}$. The

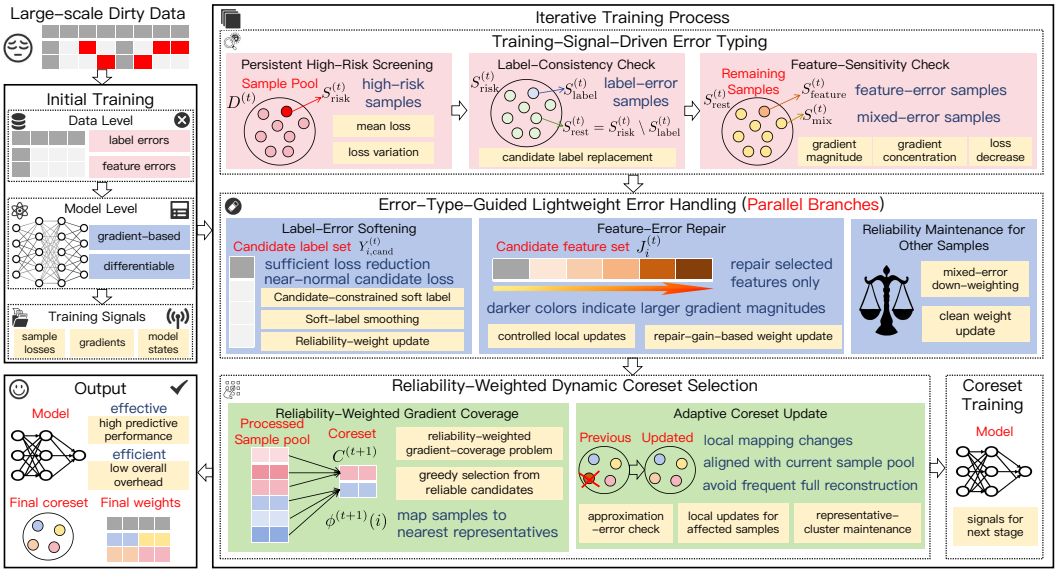


Figure 2: Overall framework of CleanCore.

mapping $\phi^{(t+1)}(i) = j$ assigns each sample $z_i^{(t+1)}$ to its coreset representative $z_j^{(t+1)}$, where j retains that representative's index in the processed sample pool.

At each stage, CleanCore trains for several epochs using only the current weighted coreset and feeds new training signals into the next stage, continuously updating sample reliability and maintaining coreset representativeness.

3 Method Design

This section presents the design of CleanCore and introduces the three components, ErrorTyping, ErrorHandle, and CoreSelect, in sequence. The overall procedure and related figures and pseudocode are provided in Appendix D; detailed calculations, design rationales, reliability and complexity analyses, and related theorems and proofs for the three components are provided in Appendices D.1, D.2, and D.3, respectively; overall reliability, training complexity, and adaptive threshold settings are provided in Appendix D.4. Within the component descriptions below, bold internal mechanism names correspond to the internal headings in the respective appendices, which provide detailed calculations and explanations.

3.1 Training-Signal-Driven Error Typing

ErrorTyping screens high-risk samples from the stage-t sample pool $D^{(t)} = \{z_i^{(t)} = (x_i^{(t)}, y_i^{(t)})\}_{i=1}^{n^{(t)}}$, determines error types through label-consistency and feature-sensitivity checks, and provides error sets and intermediate information $\mathcal{R}^{(t)}$ to ErrorHandle. Decision thresholds are collectively denoted by $\tau^{(t)}$.

(1) **Persistent High-Risk Screening.** Let r be the global training epoch at which identification is triggered in the current stage and θ_k the parameter snapshot after epoch k, with $\theta_r = \theta^{(t)}$. Within the window $W_r = \{r - L + 1, \ldots, r\}$ of the most recent L epochs, screen using the mean $\mu_i^{(t)}$ and standard deviation $\sigma_i^{(t)}$ of losses $\ell_{i,k}^{(t)} = \ell(f_{\theta_k}(x_i^{(t)}), y_i^{(t)})$: $S_{\text{risk}}^{(t)} = \left\{ z_i^{(t)} \in D^{(t)} \mid \mu_i^{(t)} \geq \tau_\mu^{(t)},\ \sigma_i^{(t)} \leq \tau_\sigma^{(t)} \right\}$. Here, $\tau_\mu^{(t)}$ and $\tau_\sigma^{(t)}$ are the mean-loss and variability thresholds, respectively, used to screen persistently high-loss, low-variability samples and reduce mistaken inclusion of hard but correct samples due to transient high losses or training fluctuations.

(2) **Label-Consistency Check.** For samples in $S_{\text{risk}}^{(t)}$, fix features and virtually replace the label with $c \in Y$. Denote the replacement loss under snapshot k by $\ell_{i,k}^{(t)}(c)$ and its window mean by $\mu_{i,c}^{(t)}$,

retaining candidate labels satisfying $Y_{i,\mathrm{cand}}^{(t)} = \left\{ c \in Y \mid \mu_i^{(t)} - \mu_{i,c}^{(t)} \geq \tau_{\mathrm{rel}}^{(t)},\ \mu_{i,c}^{(t)} \leq \tau_{\mathrm{clean}}^{(t)} \right\}$. The two thresholds respectively control the minimum loss reduction and the upper limit on replacement loss near the current normal-sample level. Samples with nonempty candidate sets form the label-error set $S_{\mathrm{label}}^{(t)}$; candidate labels and mean losses are retained for soft-label construction, avoiding premature hard relabeling. The remaining samples $S_{\mathrm{rest}}^{(t)} = S_{\mathrm{risk}}^{(t)} \setminus S_{\mathrm{label}}^{(t)}$ proceed to the next step.

(3) **Feature-Sensitivity Check.** For samples in $S_{\mathrm{rest}}^{(t)}$, denote the loss gradient with respect to input features under snapshot k and the current label by $q_{i,k}^{(t)}$ and its window mean by $\bar{q}_i^{(t)}$; $x_{i,j}^{(t)}$ and $\bar{q}_{i,j}^{(t)}$ denote the j-th components of the feature vector and mean gradient, respectively. Using the gradient-magnitude threshold $\tau_A^{(t)}$ and cumulative gradient-proportion threshold $\tau_{\mathrm{conc}}^{(t)}$, check whether the mean gradient is sufficiently large and concentrated on a few features, obtaining the candidate feature set $J_i^{(t)}$ with size at most m_0. For samples passing these checks, fix the label and perform one small-step virtual update on candidate dimensions based on the mean gradient, computing the window mean loss reduction $\Delta_i^{(t)}$ without modifying the sample pool. Samples whose gains reach the loss-reduction threshold $\tau_{\mathrm{feat}}^{(t)}$, namely $\Delta_i^{(t)} \geq \tau_{\mathrm{feat}}^{(t)}$, form the feature-error set $S_{\mathrm{feature}}^{(t)}$; the remaining samples are temporarily assigned to the mixed-error set $S_{\mathrm{mix}}^{(t)} = S_{\mathrm{rest}}^{(t)} \setminus S_{\mathrm{feature}}^{(t)}$ for conservative handling. $J_i^{(t)}$, $\bar{q}_i^{(t)}$, and $\Delta_i^{(t)}$ are used for subsequent feature repair.

All three error sets can be adjusted with new training signals; accurate and complete identification of all errors in a single iteration is not required. The typing reliability analysis is provided in Theorem 1 in Appendix D.1.

3.2 Error-Type-Guided Lightweight Error Handling

ErrorHandle applies differentiated lightweight handling according to the error sets and intermediate information from ErrorTyping and maintains reliability weights $w_i^{(t)} \in [0, 1]$. These weights characterize sample trustworthiness at the current training stage and regulate training contributions, rather than precisely measuring true data quality.

(1) **Label-Error Softening (LES).** For samples in $S_{\mathrm{label}}^{(t)}$, LES performs **instantaneous soft-label construction based on the candidate set**, using the candidate label set $Y_{i,\mathrm{cand}}^{(t)}$ and mean losses $\mu_{i,c}^{(t)}$ from Section 3.1 to assign higher probabilities to lower-loss candidates and form the instantaneous distribution $\widehat{p}_i^{(t)}(c)$. Probabilities outside the candidate set are zero to limit irrelevant-class interference while preserving uncertainty among candidates.

To reduce the effects of training fluctuations, LES performs **soft-label smoothing as training evolves**, combining the instantaneous distribution with the historical distribution $p_i^{(t)}(c)$: $p_i^{(t+1)}(c) = (1-\beta)p_i^{(t)}(c) + \beta\widehat{p}_i^{(t)}(c),\ c \in Y$, where $0 < \beta < 1$ controls the mixture of current and historical judgments. On first identification, the historical distribution is initialized with the one-hot distribution corresponding to the original label. Subsequent training and ErrorTyping both compute the probability-weighted sum of class-specific losses using the updated class probabilities, treating hard labels as one-hot distributions.

In **confidence-based lightweight weight updating**, the maximum class probability characterizes correction confidence, and linear updating assigns higher weights to samples with higher confidence. If the same dominant class maintains high confidence over multiple consecutive stages, the soft label may be converted to that class; otherwise, the soft label and corresponding weight are retained.

(2) **Feature-Error Repair (FER).** For samples in $S_{\mathrm{feature}}^{(t)}$, FER uses the candidate feature set $J_i^{(t)}$ from Section 3.1 to determine repair dimensions and controls the step size using the mean feature gradient $\bar{q}_i^{(t)}$ and virtual-update gain $\Delta_i^{(t)}$. Repair fixes the model parameters $\theta^{(t)}$ at identification and the label $y_i^{(t)}$, updating only features; continuous features use numerical representations, integer features use continuous relaxations, and categorical features use continuous relaxations of one-hot encodings.

For **lightweight iterative repair on candidate features**, let u be the local repair step, with current features initialized as $x_i^{(t,0)} = x_i^{(t)}$. Each step retains only the instantaneous input-gradient components on $J_i^{(t)}$, yielding the truncated gradient $\widetilde{q}_i^{(t,u)}$, determines update coefficients $a_{i,j}^{(t,u)}$ from relative current gradient magnitudes, and determines a capped sample-specific step size $\eta_i^{(t)}$ from the virtual-update gain and mean-gradient scale. The update is

$$x_{i,j}^{(t,u+1)} = x_{i,j}^{(t,u)} - \eta_i^{(t)} a_{i,j}^{(t,u)} \widetilde{q}_{i,j}^{(t,u)}.$$

Only candidate dimensions are updated, and bounded features are clipped to valid ranges; integer and categorical features retain continuous relaxations without rediscretization. Each step recomputes the loss, stopping when successive gains are limited or the step limit is reached. The lowest-loss version, including the initial state in the comparison, is selected as $x_i^{(t+1)}$; without effective improvement, the original features are retained.

Then, **repair-gain-driven weight maintenance** computes repair confidence from the loss reduction before and after repair and smoothly updates the sample weight toward this confidence, adjusting subsequent training contributions according to actual repair effectiveness.

(3) **Reliability Maintenance for Other Samples.** For samples in $S_{\text{mix}}^{(t)}$, mixed-error down-weighting (MDW) multiplicatively reduces weights while leaving features and labels unchanged for continued observation; samples not currently identified as erroneous receive small weight increases capped at 1 (CWU). Temporarily misclassified hard but correct samples can gradually recover their weights if they are not removed and are no longer identified as erroneous later.

After all weight updates, low-reliability sample removal (LRS) removes only samples in the current high-risk set $S_{\text{risk}}^{(t)}$ whose weights fall below the threshold $w_{\min}$, so that they no longer participate in subsequent coreset selection. The analysis of the bound on error-handling bias is provided in Theorem 2 in Appendix D.2.

3.3 Reliability-Weighted Dynamic Coreset Selection

CoreSelect builds or updates the coreset from the pool $D^{(t+1)}$ and reliability weights $w^{(t+1)}$ produced by ErrorHandle. It selects representatives through reliability-weighted gradient coverage and applies hierarchical updates as representativeness changes.

(1) **Reliability-Weighted Gradient Coverage.** To unify hard- and soft-label samples, compute the label-expected gradient under the current parameters $\theta^{(t)}$: $\bar{g}_i^{(t+1)} = \sum_{c \in Y} p_i^{(t+1)}(c) \nabla_\theta \ell \left(f_{\theta^{(t)}} \left(x_i^{(t+1)} \right), c \right)$. Hard labels are viewed as one-hot distributions. Soft labels capture label uncertainty, while reliability weights further reflect training contributions after label correction, feature repair, and mixed-risk suppression. CoreSelect screens representative candidates using the reliability threshold τ_{rep} and supplements them according to reliability when fewer than K candidates are available, obtaining the candidate set $P^{(t+1)}$. All samples in the processed pool contribute to coverage-error calculation, but only candidates may become representatives. CoreSelect selects at most K representatives by minimizing the following **reliability-weighted gradient coverage objective**:

$$F(C) = \sum_{i=1}^{n^{(t+1)}} w_i^{(t+1)} \min_{z_j^{(t+1)} \in C} \|\bar{g}_i^{(t+1)} - \bar{g}_j^{(t+1)}\|_2,$$

$$C^{(t+1)} = \operatorname*{arg\,min}_{C \subseteq P^{(t+1)},\, 1 \le |C| \le K} F(C).$$

This objective emphasizes coverage of highly reliable samples and controls coreset gradient-approximation error. **Greedy construction over highly reliable candidates** starts with the sample minimizing coverage error, then adds the candidate that decreases $F(C)$ most each round until budget K is reached or no candidates remain. To reduce overhead, candidate batches may be sampled and evaluated to select the one with the greatest gain. The nearest-representative mapping $\phi^{(t+1)}$ is updated as representatives are added, optionally using a tree index within each batch to accelerate distance queries. The monotone submodular coverage gain and approximation guarantee for pointwise greedy selection are given in Theorem 3 in Appendix D.3.

(2) **Adaptive Coreset Update.** Error handling changes labels, features, and weights, but rebuilding every stage is unnecessary. For the **global validity check**, compare the weighted gradient of the processed sample pool with that of the previous coreset under the current model. Let $G_{\text{full}}^{(t+1)}$ and $G_{\text{sub}}^{(t)}$ denote these two gradients, respectively. The relative approximation error is

$$E^{(t+1)} = \frac{\|G_{\text{full}}^{(t+1)} - G_{\text{sub}}^{(t)}\|_2}{\|G_{\text{full}}^{(t+1)}\|_2}.$$

Rebuild when no reusable coreset exists or $E^{(t+1)} > \tau_{\text{reb}}$; perform only local updates when $E^{(t+1)} \leq \tau_{\text{upd}}$; and additionally maintain representative-cluster reliability after local updates when $\tau_{\text{upd}} < E^{(t+1)} \leq \tau_{\text{reb}}$. Here, τ_{upd} and τ_{reb} are the local-update and rebuilding thresholds.

Local updating of affected samples first processes representatives: remove those deleted from the pool or no longer eligible, preferentially seeking highly reliable replacements within their former clusters; temporarily retain eligible representatives with updated gradients. Then process affected nonrepresentatives still in the pool: directly remap those whose representatives were removed or replaced. Otherwise, remap to the nearest representative in the effective coreset only when the reliability-weighted gradient deviation from the original representative exceeds the mapping-update threshold τ_{map}; retain the original mapping with the new weight otherwise.

Representative-cluster reliability maintenance checks the within-cluster reliability-weighted mean deviation. Retain the representative if it does not exceed the stability threshold τ_{clu}; otherwise, find an eligible cluster member minimizing weighted gradient distances as a replacement and map the original cluster members to it. After all necessary updates, aggregate weights using the final mapping: $\widetilde{w}_j^{(t+1)} = \sum_{i:\phi^{(t+1)}(i)=j} w_i^{(t+1)}$. This weight represents aggregated reliable training contributions rather than sample counts. The outputs are $C^{(t+1)}$, $\widetilde{w}^{(t+1)}$, and $\phi^{(t+1)}$; subsequent model updates use only this weighted coreset.

4 Experiments

4.1 Experimental Design

(1) **Datasets.** We select eleven public tabular classification datasets, covering different scales, feature types, class counts, and class distributions. Ten come from the UCI Machine Learning Repository for artificial-error injection experiments; SmartFactory comes from the REIN benchmark (Abdelaal et al., 2023) for real-world dirty-data validation. Dataset statistics are provided in Table 2 in Appendix E.1.

(2) **Baseline Methods.** We compare CleanCore with eleven representative baselines: data cleaning methods Cleanlab (Northcutt et al., 2021), MisDetect (Deng et al., 2024), and CTRL (Yue & Jha, 2024); coreset selection methods Herding (Welling, 2009), DeepFool (Moosavi-Dezfooli et al., 2016; Ducoffe & Precioso, 2018), GradMatch (Killamsetty et al., 2021), and CoreTab (Hadar et al., 2024); hybrid methods GoodCore (Chai et al., 2023) and Deem (Deng et al., 2025); and sequential methods CoreTab–MisDetect (C-M) and MisDetect–CoreTab (M-C), corresponding to "selection before cleaning" and "cleaning before selection", respectively.

(3) **Experimental Settings.** The main comparison uses MLP and tabular ResNet, while other analyses use MLP by default. On the same dataset with the same model, methods use identical preprocessing and shared training configurations; each analysis varies only the factor under study, with special settings stated separately. Artificial errors are injected only into training sets, covering label, feature, and mixed errors under random and distribution-shift settings; SmartFactory uses dirty data for training and clean data for testing.

(4) **Evaluation Metrics.** We use Accuracy, Macro-F1 (hereafter F1), and runtime to measure predictive performance and runtime efficiency.

Dataset details, baseline descriptions, experimental settings including training configurations, data splitting, error injection, and software and hardware environments, and metric definitions and calculations are provided in parts (1)–(4) of Appendix E.1, respectively.

Table 1: Predictive performance and runtime with MLP. CleanCore is our proposed method. ACC and F1 are percentages; TIME is in seconds. Darker and lighter blue backgrounds indicate the best and second-best distinct values in each dataset column, respectively; best values are bold, second-best values are underlined, and ties share the same formatting.

ACC (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Cleanlab	98.26	**100.00**	92.46	96.20	80.81	76.78	92.99	88.39	98.74	80.14	79.86
MisDetect	97.39	**100.00**	92.26	96.48	85.18	77.08	92.98	88.49	99.51	78.24	79.98
CTRL	98.26	**100.00**	92.46	95.40	85.02	76.00	93.11	88.70	99.32	82.28	**80.40**
Herding	73.91	91.64	91.10	85.19	76.22	76.78	81.47	81.04	86.88	71.13	62.56
DeepFool	86.96	92.00	91.68	93.34	74.64	74.73	63.13	74.39	85.90	65.10	67.29
GradMatch	94.78	98.18	92.07	95.91	80.76	76.02	91.87	84.15	98.68	77.66	76.57
CoreTab	96.52	92.73	90.52	88.39	82.05	74.77	87.44	88.42	99.21	78.26	78.41
GoodCore	90.43	94.18	90.91	89.54	81.37	76.02	87.29	86.42	99.65	79.85	79.72
Deem	94.78	98.18	91.49	94.60	83.65	75.62	71.60	86.77	98.82	72.72	79.98
C-M	94.78	93.82	92.26	86.82	78.87	75.94	88.53	83.85	92.73	72.43	78.18
M-C	96.52	94.18	92.46	93.22	80.95	77.27	93.40	88.86	99.33	78.97	78.50
CleanCore	**99.13**	**100.00**	**93.42**	**96.66**	**85.52**	**77.75**	**94.53**	**90.04**	**99.73**	**82.48**	80.09

F1 (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Cleanlab	98.14	**100.00**	54.69	96.23	75.60	50.66	92.98	84.97	98.12	61.53	79.42
MisDetect	97.20	**100.00**	50.36	96.47	82.20	53.11	92.92	84.85	99.26	48.33	79.61
CTRL	98.14	**100.00**	52.68	95.35	82.81	53.70	93.01	85.95	98.98	**69.92**	**79.97**
Herding	73.90	91.52	47.67	84.25	74.14	52.50	81.54	75.58	74.57	44.95	62.52
DeepFool	84.95	91.85	50.05	93.36	66.19	55.69	63.01	52.61	70.20	36.55	62.75
GradMatch	94.31	98.17	47.94	95.93	77.75	43.19	91.74	77.77	97.98	56.99	76.01
CoreTab	96.32	92.68	49.47	88.37	79.15	56.48	87.35	84.82	98.82	55.15	78.35
GoodCore	90.07	94.13	56.37	89.72	79.06	57.43	87.14	81.05	99.48	55.37	79.30
Deem	94.43	98.17	47.78	94.67	80.40	53.54	70.62	82.46	98.25	46.19	79.54
C-M	94.31	93.76	54.50	86.88	77.01	56.23	88.50	78.21	88.29	47.53	77.39
M-C	96.21	94.13	48.04	93.24	77.67	51.15	93.39	85.58	98.99	63.02	77.70
CleanCore	**99.07**	**100.00**	**59.65**	**96.71**	**83.43**	**57.54**	**94.52**	**87.65**	**99.59**	69.70	79.65

TIME (s) ↓

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Cleanlab	1.99	8.98	19.36	34.36	23.31	49.33	114.25	203.91	382.90	668.89	5239.03
MisDetect	1.62	7.37	16.34	33.08	35.49	42.96	89.46	210.21	217.09	651.20	4922.27
CTRL	2.12	6.56	22.44	25.23	55.53	51.47	82.48	170.30	313.98	629.99	7763.39
Herding	0.81	1.02	2.32	4.19	5.28	5.72	**6.44**	27.64	26.53	205.50	652.32
DeepFool	0.98	1.76	2.99	4.58	10.17	8.18	21.10	46.56	53.88	222.90	756.62
GradMatch	1.07	0.83	2.29	1.76	2.87	6.04	15.00	16.56	16.43	148.00	387.84
CoreTab	0.87	1.22	2.45	4.61	4.54	5.83	15.69	20.30	33.53	143.22	396.48
GoodCore	0.85	1.13	2.42	4.09	5.16	5.93	25.12	41.21	51.35	281.59	799.09
Deem	**0.51**	**0.66**	2.46	3.33	2.81	5.29	11.57	18.32	27.73	155.63	568.74
C-M	1.01	1.34	2.66	7.03	8.04	6.50	24.97	32.85	55.50	225.82	495.06
M-C	1.26	2.65	4.20	11.38	13.79	19.29	50.54	80.73	131.39	348.40	940.18
CleanCore	0.80	0.73	**2.11**	**1.61**	**2.68**	**5.10**	11.85	**16.17**	**16.16**	**127.11**	**359.09**

4.2 Overall Evaluation

(1) **Comparison under Fixed Noise.** We compare methods on eleven datasets using MLP and tabular ResNet. For the ten clean reference datasets, 20% random errors are injected into the training sets; SmartFactory uses its existing dirty training set and clean test set. Coreset budgets are set to 10%, 5%, or 1% depending on the dataset. All selection methods and both downstream models use the same budget on a given dataset, and subsequent experiments retain these settings by default.

Table 1 reports the MLP results, while Table 3 in Appendix E.2 (1) reports the tabular ResNet results. **With MLP, CleanCore achieves the best ACC on ten datasets and the best F1 on nine; with tabular ResNet, its ACC and F1 are best on all eleven datasets.** On the real-world dirty dataset SmartFactory and on Seismic Bumps, where mixed feature types and marked class imbalance coexist, CleanCore achieves the best ACC and F1 under both models. Compared with GradMatch, which also uses gradients for selection, CleanCore achieves higher ACC and F1 on all datasets under both models, supporting the effectiveness of considering data quality in coreset selection.

For runtime efficiency, CleanCore achieves approximately 2–24$\times$ speedups over the three cleaning methods on all datasets under both models. Coreset training reduces the samples accessed in

each model-update epoch, while candidate-batch sampling, tree-based mapping, and adaptive updating control construction and maintenance costs. Overall, **CleanCore achieves speedups with predictive performance close to or better than cleaning methods, and obtains better predictive performance with runtime efficiency close to or better than coreset methods, demonstrating the strongest overall performance.** Specific budget settings and detailed comparative analysis are provided in Appendix E.2 (1), and further time composition analysis is provided in Appendix E.2 (3).

(2) **Comparison across Noise Intensities and Types.** We compare the predictive performance of GradMatch, Deem, and CleanCore under different error intensities and types. All three use gradient information to construct or update coresets, but respectively do not handle errors, mainly handle label errors, and simultaneously detect and handle label, feature, and mixed errors. Experiments are conducted on six datasets, with random or distribution-shift errors injected into training sets at proportions ranging from 6% to 30%.

CleanCore achieves the best ACC and F1 on all six datasets across both error types and different proportions. As the error proportion increases, GradMatch and Deem decline more noticeably on some datasets, while CleanCore maintains high predictive performance overall, with a more pronounced advantage on the more challenging Covertype dataset. This shows that CleanCore reduces the negative effects of different errors on model training and maintains good predictive performance as error type and intensity change. Specific experimental settings, Figure 7, Table 4, and detailed result analysis are provided in Appendix E.2 (2).

4.3 Hyperparameter Analysis

This subsection analyzes the effects of hyperparameters on predictive performance and runtime efficiency. All three experiments use Pendigits, Magic, Sensorless, MiniBooNE, and Covertype, with 20% random errors injected into training sets. Specific settings, Figure 8, and detailed analyses for coreset budget and sliding window length are provided in Appendix E.3 (1)–(2); the initial-training-epoch experiment is provided in Appendix E.3 (3).

(1) **Coreset Budget.** Accuracy generally increases with the budget on all five datasets, improving by 0.88–4.33 percentage points from the lowest to highest budget, indicating that a larger coreset improves full-data coverage. Runtime also generally increases, but overall efficiency remains high, so the coreset size can be selected according to predictive-performance and efficiency requirements.

(2) **Sliding Window Length.** As the sliding window length L varies from 2 to 10, Accuracy varies by only 0.10–0.83 percentage points across the five datasets, indicating insensitivity to window length. Lengths from $L = 4$ to $L = 6$ usually achieve high predictive performance. Runtimes are generally similar across window lengths, and the additional overhead of window statistics is small and largely stable.

The complete ablation experiments are provided in Appendix E.4; analyses of the effectiveness of ErrorTyping and ErrorHandle and threshold sensitivity are provided in parts (1)–(3) of Appendix E.5, respectively. The random sampling control experiment is provided in Appendix E.5 (4).

5 Conclusion

We propose CleanCore, a data-quality-aware coreset selection method for training on large-scale dirty tabular data with multiple error types. Through staged iteration, it combines training-signal-driven error typing, error-type-guided lightweight error handling, and reliability-weighted dynamic coreset selection, maintaining representativeness while reducing error interference and model training costs. Theoretical analysis provides support in terms of typing reliability, error-handling bias, and coreset representativeness. Comparative experiments on eleven public tabular datasets using MLP and tabular ResNet show that CleanCore **balances predictive performance and runtime efficiency**, achieves approximately 2–$24\times$ speedups over three data cleaning methods, and performs well with real-world dirty data and different injected-error conditions, demonstrating the strongest overall performance. Further experiments validate its stability and the effectiveness and efficiency of its main mechanisms. The method provides a unified approach to reliable and efficient training on large-scale dirty tabular data.

AI USE STATEMENT

During the writing of this paper, generative AI tools were used to assist with literature search, manuscript translation, language polishing, adjustments to the structure of the paper, and LaTeX typesetting. The authors are responsible for verifying the accuracy of AI-assisted content and take responsibility for the text, citations, research conclusions, and related materials in the final version of the paper.

REPRODUCIBILITY STATEMENT

To support the reproducibility of the research results, the main text presents the problem definition and main methodological procedure of CleanCore. Appendix D provides algorithm pseudocode, detailed calculations, theoretical assumptions, and proofs; Appendix E provides dataset descriptions, baseline methods, detailed experimental settings, evaluation metric calculations, and supplementary experimental results. We plan to release the relevant code and instructions for running it after the paper is accepted to further support the reproducibility of the research results.

REFERENCES

Mohamed Abdelaal, Christian Hammacher, and Harald Schoening. REIN: A comprehensive benchmark framework for data cleaning methods in ML pipelines. *arXiv preprint arXiv:2302.04702*, 2023.

E. Alpaydin and Fevzi. Alimoglu. Pen-Based Recognition of Handwritten Digits. UCI Machine Learning Repository, 1996. DOI: https://doi.org/10.24432/C5MG6K.

Devansh Arpit, Stanisław Jastrzębski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, et al. A closer look at memorization in deep networks. In *International conference on machine learning*, pp. 233–242. PMLR, 2017.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Martyna Bator. Dataset for Sensorless Drive Diagnosis. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C5VP5F.

Rajen Bhatt and Abhinav Dhall. Skin Segmentation. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C5T30C.

Jock Blackard. Covertype. UCI Machine Learning Repository, 1998. DOI: https://doi.org/10.24432/C50K5N.

R. Bock. MAGIC Gamma Telescope. UCI Machine Learning Repository, 2004. DOI: https://doi.org/10.24432/C52C8B.

Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. Deep neural networks and tabular data: A survey. *IEEE transactions on neural networks and learning systems*, 35(6):7499–7519, 2022.

Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM review*, 60(2):223–311, 2018.

Chengliang Chai, Jiabin Liu, Nan Tang, Ju Fan, Dongjing Miao, Jiayi Wang, Yuyu Luo, and Guoliang Li. GoodCore: Data-effective and data-efficient machine learning through coreset selection over incomplete data. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on management of data*, pp. 2201–2206, 2016.

Yuhao Deng, Chengliang Chai, Lei Cao, Nan Tang, Jiayi Wang, Ju Fan, Ye Yuan, and Guoren Wang. MisDetect: Iterative mislabel detection using early loss. Association for Computing Machinery (ACM), 2024.

Yuhao Deng, Chengliang Chai, Kaisen Jin, Linan Zheng, Lei Cao, Ye Yuan, and Guoren Wang. Two birds with one stone: Efficient deep learning over mislabeled data through subset selection. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.

Melanie Ducoffe and Frederic Precioso. Adversarial active learning for deep networks: a margin based approach. *arXiv preprint arXiv:1802.09841*, 2018.

Tlamelo Emmanuel, Thabiso Maupong, Dimane Mpoeleng, Thabo Semong, Banyatsang Mphago, and Oteng Tabona. A survey on missing data in machine learning. *Journal of Big data*, 8(1):140, 2021.

Dan Feldman. Core-sets: An updated survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 10(1):e1335, 2020.

Benoît Frénay and Michel Verleysen. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5):845–869, 2013.

Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3 (3):209–226, 1977.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323. JMLR Workshop and Conference Proceedings, 2011.

Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. Revisiting deep learning models for tabular data. *Advances in neural information processing systems*, 34:18932–18943, 2021.

Aviv Hadar, Tova Milo, and Kathy Razmadze. Datamap-driven tabular coreset selection for classifier training. *Proceedings of the VLDB Endowment*, 18(3):876–888, 2024.

Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE intelligent systems*, 24(2):8–12, 2009.

Ihab F Ilyas and Xu Chu. *Data cleaning*. Morgan & Claypool, 2019.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. pmlr, 2015.

Krishnateja Killamsetty, Sivasubramanian Durga, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. GRAD-MATCH: Gradient matching based data subset selection for efficient deep model training. In *International Conference on Machine Learning*, pp. 5464–5474. PMLR, 2021.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Volker Lohweg. Banknote Authentication. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C55P57.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2574–2582, 2016.

George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—I. *Mathematical programming*, 14(1):265–294, 1978.

Curtis Northcutt, Lu Jiang, and Isaac Chuang. Confident learning: Estimating uncertainty in dataset labels. *Journal of Artificial Intelligence Research*, 70:1373–1411, 2021.

Juri Opitz and Sebastian Burst. Macro F1 and macro F1. *arXiv preprint arXiv:1911.03347*, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Jeff M Phillips. Coresets and sketches. *arXiv preprint arXiv:1601.00617*, 2016.

Erhard Rahm, Hong Hai Do, et al. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

Byron Roe. MiniBooNE particle identification. UCI Machine Learning Repository, 2005. DOI: https://doi.org/10.24432/C5QC87.

Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, 2020.

Marek Sikora and Lukasz Wrobel. seismic-bumps. UCI Machine Learning Repository, 2010. DOI: https://doi.org/10.24432/C5W902.

Hwanjun Song, Minseok Kim, Dongmin Park, Yooju Shin, and Jae-Gil Lee. Learning from noisy labels with deep neural networks: A survey. *IEEE transactions on neural networks and learning systems*, 34(11):8135–8153, 2022.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

W Nick Street, William H Wolberg, and Olvi L Mangasarian. Nuclear feature extraction for breast tumor diagnosis. In *Biomedical image processing and biomedical visualization*, volume 1905, pp. 861–870. SPIE, 1993.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 3645–3650, 2019.

Swabha Swayamdipta, Roy Schwartz, Nicholas Lourie, Yizhong Wang, Hannaneh Hajishirzi, Noah A Smith, and Yejin Choi. Dataset cartography: Mapping and diagnosing datasets with training dynamics. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 9275–9293, 2020.

Max Welling. Herding dynamical weights to learn. In *Proceedings of the 26th annual international conference on machine learning*, pp. 1121–1128, 2009.

Daniel Whiteson. SUSY. UCI Machine Learning Repository, 2014. DOI: https://doi.org/10.24432/C54606.

Chang Yue and Niraj K Jha. CTRL: Clustering training losses for label error detection. *IEEE Transactions on Artificial Intelligence*, 5(8):4121–4135, 2024.

Daochen Zha, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu. Data-centric artificial intelligence: A survey. *ACM Computing Surveys*, 57(5):1–42, 2025.

A Related Work

Data quality and data-centric machine learning. Data quality and training-data scale jointly affect the predictive performance and computational cost of machine learning. Zha et al. (2025) review data-centric AI from the perspectives of training data development, inference data development, and data maintenance, highlighting the role of data in building AI systems. For different data quality problems, Emmanuel et al. (2021) review missing-data handling techniques in machine learning, while Song et al. (2022) systematically discuss learning with noisy labels using deep neural networks. These surveys provide background for understanding data quality problems and their treatment. REIN (Abdelaal et al., 2023) provides a benchmark framework for studying the effects of data cleaning on downstream machine learning. We use its SmartFactory data under the unified experimental settings of this paper to evaluate predictive performance and overall runtime efficiency on real-world dirty data.

Data cleaning for machine learning. Model predictions and training behavior provide information for identifying erroneous data. Confident learning, implemented in Cleanlab (Northcutt et al., 2021), uses predicted probabilities to estimate the relationship between observed and latent true labels and identify potential label issues. MisDetect (Deng et al., 2024) iteratively detects incorrect labels using early training losses and incorporates influence-based verification to improve detection accuracy. CTRL (Yue & Jha, 2024) clusters sample training-loss curves to distinguish correctly labeled and mislabeled samples. These methods assess label quality through prediction confidence, early losses, and loss dynamics, respectively. CleanCore further considers tabular data with coexisting label, feature, and mixed errors. It combines recent loss statistics, label replacement, and feature-gradient signals for error typing, and uses sample reliability after differentiated handling for coreset selection and subsequent training.

Coreset selection. Coreset selection retains a small representative training subset to reduce sample access and computation during model updates (Feldman, 2020). Different methods assess representativeness through optimization directions or sample learning behavior. GradMatch (Killamsetty et al., 2021) selects weighted subsets to match the gradients of the training or validation set and uses orthogonal matching pursuit to solve the selection problem. CoreTab (Hadar et al., 2024) constructs data maps (Swayamdipta et al., 2020) from the training behavior of gradient boosting decision trees for tabular classification. It distinguishes regions with different learning difficulties and retains samples from difficult regions together with representatives from easy regions. These methods illustrate the roles of gradient approximation and training-behavior analysis in representative data selection. Since errors may affect these selection signals, **CleanCore couples error-type-guided handling with coreset selection**: post-handling sample reliability guides gradient coverage and representative eligibility, while changes in labels, features, and reliability drive adaptive coreset maintenance. When evaluating the overall efficiency of coreset training, we account for training costs as well as coreset selection and maintenance costs.

Joint consideration of data quality and coreset selection. Existing studies integrate data quality improvement and training-data reduction into a unified process. GoodCore (Chai et al., 2023) addresses incomplete data by representing possible repair combinations as possible worlds, selecting coresets through gradient approximation under uncertainty, and incorporating imputation strategies to improve selection efficiency and quality. Deem (Deng et al., 2025) addresses mislabeled data by using training losses and gradients to estimate label uncertainty. It selects representative subsets through gradient approximation under soft labels and dynamically adjusts them during training. These studies have explored quality-aware data selection. CleanCore further addresses the coexistence of multiple error types considered in this paper by connecting error typing, type-guided lightweight handling, reliability updating, and dynamic coreset selection. Post-handling reliability contributes to representativeness assessment, while new signals from coreset training inform subsequent quality judgments. We evaluate this iterative design in terms of predictive performance and overall runtime efficiency through comparisons with related methods and sequential combinations of cleaning and selection.

B SUPPLEMENTARY EXPLANATIONS FOR THE INTRODUCTION

This section supplements the research background and example analysis, further explaining the method's design motivation, main challenges and contributions, scope of applicability, and preprocessing of explicit errors.

Data scale and quality. In modern deep learning, data provide an essential foundation for models to acquire knowledge. Training complex models on large numbers of instances has become common practice. The value of data for training depends on both their scale and quality. Errors often arise during the collection, transmission, and manual annotation of real-world data (Rahm et al., 2000; Emmanuel et al., 2021; Song et al., 2022).

Error types. According to the affected object, label errors occur when sample labels differ from their true labels (Frénay & Verleysen, 2013), feature errors occur when sample features differ from their true values, and mixed errors affect both labels and features. According to detectability, explicit errors include missing values, out-of-range values, and invalid types that can be directly detected through data constraints; implicit errors include incorrect labels and anomalous values within legal ranges that appear valid but are inconsistent with the true semantics. These two classifications describe the objects affected by errors and their detectability, respectively.

The role of coresets. A high-quality, representative coreset could preserve useful information, reduce interference from low-quality samples, and lower training costs by reducing sample access, thereby enabling more reliable and efficient training while maintaining predictive performance.

Representative methods. Data cleaning methods, represented by Cleanlab (Northcutt et al., 2021), improve the reliability of training data by identifying and handling low-quality samples. Coreset selection methods, represented by GradMatch (Killamsetty et al., 2021), select a representative subset to approximate full-data training and reduce training costs. Hybrid methods, such as GoodCore (Chai et al., 2023) and Deem (Deng et al., 2025), handle specific error types while constructing representative coresets.

Quality problems in transaction records. Actual transaction records often contain substantial quality problems, including label errors such as mislabeling and missing labels, and feature errors such as missing features, out-of-range values, invalid data types, and anomalies within legal ranges. In Figure 1, missing and out-of-range values are explicit errors, whereas incorrect labels and x_2 in t_3 are implicit errors. These errors may interfere with the model's learning of fraud patterns, causing normal transactions to be classified as fraudulent or fraudulent transactions to be missed, resulting in financial losses.

How the three categories perform. Case c_2 in Figure 1 corresponds to data cleaning methods. These methods perform additional error identification and handling on millions of transaction records beyond model training. Although this reduces the influence of erroneous data, separate full-data processing and subsequent training incur substantial overhead. Case c_3 corresponds to coreset selection methods. These methods select a small number of transaction records for training, but mislabeled or feature-corrupted records selected as representatives may interfere with the model's learning of fraud patterns. Case c_4 corresponds to hybrid methods such as GoodCore and Deem. GoodCore handles only missing values in transaction records, whereas Deem handles only label errors. When label errors and multiple feature errors coexist, handling specific error types cannot cover all low-quality records, and model performance remains affected.

Specific costs of sequential combinations and extensions. For cleaning before selection, the additional full-data processing includes error detection and repair and therefore incurs substantial processing overhead; selecting before cleaning allows errors in the original data to affect coreset selection. To extend GoodCore or Deem, candidate labels and candidate feature repairs can be considered jointly, with samples selected and updated based on weighted loss or gradient statistics. In practice, however, feature spaces are often high-dimensional and contain continuous variables. Candidate repair combinations grow rapidly with feature dimensionality, leading to prohibitive computation and memory costs that may offset the efficiency gains of coreset training.

Mechanisms underlying reliability and efficiency. In the iterative feedback process shown in c_5 of Figure 1, the model primarily updates from reliable, representative samples, reducing the interference of erroneous records with fraud-pattern learning. Coreset training reduces the number of samples participating in model updates, while error identification and handling use training signals, thereby improving overall runtime efficiency with little additional overhead.

Error identification, targeted handling, and coreset selection interact during training, and unreliability in one step may propagate to later steps and affect training outcomes.

Difficulties in error typing. Training signals such as sample losses are prone to fluctuations caused by erroneous data, and a single observation is insufficient to reliably assess sample quality. Different error types and hard but correct samples, including minority-class samples, may all exhibit high loss. Their observed behavior lacks clear boundaries, making error typing difficult. In Example 1, mislabeled transactions, anomalous transaction features, and rare but correct records may all have persistently high losses. Loss alone cannot readily distinguish the underlying causes or provide reliable types for subsequent handling.

Effectiveness and overhead of error handling. Label, feature, and mixed errors influence training differently. Even after identifying an error type, choosing an effective matching treatment remains difficult, and inappropriate handling may introduce new bias. Ensuring effective handling often requires repeated evaluations over many samples and high-dimensional feature spaces. For the transaction data in Example 1, such computation incurs substantial time and memory costs and may offset coreset training speedups. Effective, targeted handling of multiple error types at low cost is therefore a key challenge.

Bias propagation in coreset selection. Coreset selection relies on training signals to assess representativeness, but erroneous data may distort those signals and cause unreliable samples to be selected as representatives. With multiple error types in Example 1, erroneous transactions entering the coreset may interfere with the model's learning of fraud patterns. This bias can propagate to subsequent training signals, further distorting selection and reducing representativeness, eventually moving the model away from a reasonable optimization direction.

Staged iterative framework. CleanCore's staged iterative framework unifies data quality modeling and coreset selection, dynamically adjusts the coreset to the training state, and trains the model using the resulting coreset and its weights. Each stage performs suspicious-sample identification and error typing, targeted lightweight handling, and reliability-weighted coreset selection. New training signals continuously update sample reliability weights and the coreset. The process maintains reliability and representativeness as the training state changes, and, by reducing the data used for model training, reduces multi-epoch training complexity within a stage from $O(E_{\mathrm{tr}} n^{(t+1)} c_{\mathrm{tr}})$ to $O(E_{\mathrm{tr}} K c_{\mathrm{tr}})$. Since $K \ll n^{(t+1)}$, the framework substantially reduces sample access costs for each model update.

Mechanism and analysis of ErrorTyping. ErrorTyping aims to address unstable error judgments during noisy training. It uses a sliding window to compute loss statistics for current samples under recent model snapshots, combining label replacement and feature-gradient signals to distinguish label, feature, and mixed errors while seeking to reduce misidentification of hard but correct samples. This reduces the influence of single-epoch fluctuations on quality judgments. Theorem 1 proves that the conditional typing disagreement probability P_{id} has a controllable upper bound, $\Pr(d_i \le C_s \varepsilon \mid \widehat{T}_i \neq \mathrm{clean})$. Using existing model snapshots, a fixed window, and successive candidate screening controls evaluation overhead, with a main computational cost of approximately $O(n^{(t)})$.

Mechanism and analysis of ErrorHandle. ErrorHandle aims to efficiently suppress the influence of the error types identified by ErrorTyping. It applies differentiated treatments, evaluates the reliability of processed samples at the current stage, and dynamically adjusts their reliability weights. Theorem 2 bounds error-handling bias E_{handle} by R, showing that lightweight handling and reliability weighting jointly control residual perturbations. Handling primarily operates on a small candidate error set, while most other samples receive constant-cost weight updates, giving approximately $O(n^{(t)})$ complexity.

Mechanism and analysis of CoreSelect. Based on processed samples and their reliability weights, CoreSelect constructs a coreset through reliability-weighted gradient coverage to approximate the reliability-weighted training direction of the processed sample pool. Theorem 3 establishes that the coverage gain $H(C)$ is monotone and submodular, giving the greedy coreset C_g a standard approximation guarantee and bounding coreset approximation bias by $E_{\text{core}} \leq F(C_g)$, thereby providing theoretical support for the representativeness and stability of coreset training. Candidate-batch sampling and tree-based mapping can avoid pointwise greedy search over the complete candidate set and reduce construction cost from $O((n^{(t+1)})^2 K)$ to approximately $O(\frac{n^{(t+1)} K \log B}{B})$, and local adaptive updates further reduce overall overhead.

Experimental validation. Comparisons with eleven representative baselines on eleven public tabular datasets using MLP and tabular ResNet show that CleanCore achieves the best ACC and F1 on the vast majority of datasets under both models and also achieves the best performance in the real-world dirty-data setting of SmartFactory. Across all datasets, it provides approximately 2–24$\times$ runtime speedups over three data cleaning methods. CleanCore remains among the leading methods across error types and rates. Runtime composition, sensitivity, ablation, and supplementary experiments further show its stability and the effectiveness and efficiency of its main mechanisms. Overall, CleanCore combines the predictive performance of cleaning methods with the training efficiency of coreset selection and achieves the best overall performance.

Unified handling of explicit and implicit errors. The implementation handles explicit and implicit errors. Explicit errors, including missing values, out-of-range values, and invalid types, can first be located through rule checks and converted into implicit errors with legal values that may still deviate from their true semantics. They can then be handled together with existing implicit errors such as mislabels and feature anomalies within legal ranges. The subsequent method therefore primarily discusses implicit errors after this conversion.

Applicable models and training signals. CleanCore targets supervised classification on structured tabular data. It applies to deep learning classifiers optimized through gradients and primarily uses sample losses, predictions, and gradients generated during training without depending on a specific network architecture.

C Supplementary Explanations for the Overview

This section provides formal definitions of the basic concepts, supplements the problem definition and analysis, and explains CleanCore's overall framework and iterative feedback process.

C.1 Supplementary Explanations for Preliminaries and the Problem Definition

Preliminaries. This part supplements the basic concepts used in the main text and formally defines supervised learning on structured tabular data, dirty data and error types, data cleaning, and coreset selection.

We consider supervised learning on structured tabular data (Borisov et al., 2022). Each sample consists of a feature vector and a class label, and the model learns a mapping from features to labels by minimizing the training loss (Bottou et al., 2018).

Definition 2 (Supervised learning on structured tabular data)**.** *Given a structured table $D = \{(x_i, y_i)\}_{i=1}^n$ of n samples with d features, $x_i \in \mathbb{R}^d$ is the feature vector, $y_i \in Y$ is the label, Y is the class-label set, and $|Y|$ is the number of classes. The downstream model is a differentiable predictor f_θ with parameters θ, and the loss of an individual sample is $\ell_i(\theta) = \ell(f_\theta(x_i), y_i)$.*

We consider continuous features, integer-valued numerical features, and discrete categorical features commonly found in structured tabular data. Continuous features are represented numerically, integer-valued features are treated as discrete observations of continuous features, and categorical features are one-hot encoded into a common numerical input space. Different feature types can thus be handled within the same training framework.

During actual data collection and annotation, training sets often contain various types of data errors. These errors perturb loss and gradient signals and thereby affect the direction of model optimization. We therefore distinguish the following error types according to the affected object.

Definition 3 (Dirty data and error types)**.** *Let (x_i^*, y_i^*) be the true clean form of a sample. A label error occurs if $y_i \neq y_i^*$ and $x_i = x_i^*$; a feature error occurs if $x_i \neq x_i^*$ and $y_i = y_i^*$; and a mixed error occurs if both $x_i \neq x_i^*$ and $y_i \neq y_i^*$. A dataset containing at least one sample with any of these errors is called a dirty dataset.*

To reduce the influence of dirty data, data cleaning is typically needed to detect and repair errors so that the data more closely approximate their true clean form (Rahm et al., 2000; Chu et al., 2016; Ilyas & Chu, 2019).

Definition 4 (Data cleaning)**.** *Data cleaning is a mapping $\mathrm{Clean}(D) : D \longrightarrow D'$, where D' is the processed dataset. Its objective is to reduce the effects of noise and inconsistencies while preserving useful information as much as possible, thereby improving data usability and providing more stable supervision for model training.*

Training on the full dataset is expensive at scale. Coreset selection therefore approximates full-data training behavior using a representative subset of controlled size to reduce computation (Phillips, 2016).

Definition 5 (Coreset selection)**.** *Coreset selection uses the mapping $\mathrm{Select}(D, K) : D \longrightarrow C$ to select a coreset satisfying $|C| \leq K$ under budget K and assigns weights $\widetilde{w}$ to selected samples, yielding a weighted coreset $(C, \widetilde{w})$ that approximates full-data training behavior.*

Data cleaning thus focuses on improving data quality, whereas coreset selection focuses on reducing training costs. We study how to combine sample-quality assessment and coreset selection under a limited budget to balance model performance and overall runtime efficiency on large-scale data containing label, feature, and mixed errors.

Problem Definition. The currently available training pool $D = \{(x_i, y_i)\}_{i=1}^n$ contains label errors, feature errors, and mixed errors affecting both. The selected weighted coreset $(C, \widetilde{w})$ is used for subsequent model training. During training, the coreset and its weights may be adjusted as the model state and sample reliability change, and model updates continue using the adjusted weighted coreset. The objective is to reduce overall overhead while obtaining performance close to training on the full high-quality dataset. To achieve this, the coreset should remain reliable and representative as the model's training state changes.

If coreset selection is performed independently of model training, the coreset can only use limited information from the data themselves. It cannot use training signals to identify erroneous samples and select reliable samples that better support model updates. Similarly, selecting a coreset only once during training without subsequent updates may cause the original subset to gradually lose representativeness as the model state and sample reliability change. Once erroneous samples enter the coreset, their interference may also accumulate during subsequent training and move the model away from a reasonable optimization direction.

Iteratively constructing or updating the coreset during training is better aligned with these objectives. After several training epochs, newly generated training signals can be used to reassess sample reliability and representativeness, identify and handle potentially erroneous samples, and adjust the coreset and its weights accordingly. Training continues on the adjusted coreset, and the new model state supplies more information for the next adjustment. This process continuously adapts the coreset to changes in the model state and sample reliability so that updates primarily use currently highly reliable, representative samples.

Compared with the preceding two approaches, iterative construction and updating can more promptly reduce the persistent effects of erroneous data and maintain a more representative coreset, better supporting predictive performance and overall runtime efficiency.

C.2 Supplementary Explanations for the Method Framework

Framework and initial training. CleanCore unifies training-signal-driven error typing, error-type-guided lightweight error handling, and reliability-weighted dynamic coreset selection within

the same training process. Because usable training signals are initially unavailable for error identification and coreset selection, it first trains on the full dataset for several epochs to obtain preliminary losses, gradients, and model states before entering the iterative process.

Training-signal-driven error typing (ErrorTyping). ErrorTyping uses recently retained model snapshots to evaluate the current sample pool and performs persistent high-risk screening to obtain $S_{\text{risk}}^{(t)}$, reducing the search space for subsequent typing. It then checks label consistency by evaluating candidate label replacements to determine whether label errors explain the anomalies, yielding $S_{\text{label}}^{(t)}$. For remaining high-risk samples that label replacement cannot explain, feature-sensitivity checks use feature-related signals to identify feature errors. Other samples whose types remain unclear are temporarily assigned to the mixed-error candidate set.

Error-type-guided lightweight error handling (ErrorHandle). ErrorHandle applies differentiated treatments according to the typing results. Label-error samples undergo label-error softening: candidate labels form soft labels, and sample reliability weights are updated. Feature-error samples undergo feature-error repair: a small number of repairable features receive gradient-based corrections with controlled step sizes, and reliability weights are maintained after repair. Among other samples, those currently classified as mixed errors are down-weighted, while those not classified as erroneous are gradually up-weighted. After all weight updates, only samples in the current high-risk set $S_{\text{risk}}^{(t)}$ are checked; those whose updated weights fall below the removal threshold are removed from the sample pool.

Reliability-weighted dynamic coreset selection (CoreSelect). CoreSelect first constructs gradient representations of processed samples. Hard-label gradients use the current label, whereas soft-label gradients combine class-specific gradients according to their probabilities. Sample reliability weights then guide selection. As the sample distribution changes across stages, CoreSelect checks representativeness at stage boundaries and chooses between reliability-weighted gradient coverage and adaptive coreset update, maintaining representativeness of the currently corrected pool at as low a cost as possible.

The correspondence between the processed sample pool and coreset samples is $\phi^{(t+1)} : \{1, \ldots, n^{(t+1)}\} \longrightarrow \{j : z_j^{(t+1)} \in C^{(t+1)}\}$, where $n^{(t+1)}$ is the number of samples in the processed pool and $z_i^{(t+1)}$ is its i-th sample. The mapping $\phi^{(t+1)}(i) = j$ means that sample $z_i^{(t+1)}$ is represented by $z_j^{(t+1)}$ in the coreset $C^{(t+1)}$, and j retains that representative's index in the processed sample pool.

The role of iterative feedback. New signals generated by coreset training are used for error identification, handling, and coreset selection at the next stage. This iterative process continuously updates sample reliability and maintains coreset representativeness, reducing the effects of label, feature, and mixed errors and improving predictive performance and overall runtime efficiency on large-scale dirty data.

D Supplementary Materials for Method Design

This section presents CleanCore's overall procedure and the illustrations, detailed calculations, design rationales, and pseudocode for its components, together with relevant theoretical proofs, analyses of overall reliability and computational complexity, and explanations of adaptive threshold settings.

Based on staged iterative training, CleanCore unifies error identification, error handling, and coreset selection within the same iterative process. Algorithm 1 presents its overall procedure. Specifically, CleanCore first obtains usable loss and gradient signals through a small amount of initial training. At each subsequent training stage, ErrorTyping uses the current training signals to identify high-risk samples and determine their error types; ErrorHandle performs lightweight handling according to the error types and updates sample reliability weights; and CoreSelect constructs or updates the coreset based on the processed sample pool and its reliability weights. The model then continues training only on the current weighted coreset and feeds new training signals into the next stage.

Algorithm 1: CleanCore Overall Framework

Input: D (dirty dataset), M (model), coreset budget K, maximum stages $T_{\max}$
Output: $C^{(t)}$ (coreset), $\widetilde{w}^{(t)}$ (coreset weights), trained $M(\theta^{(t)})$

$C^{(0)} \leftarrow \varnothing$; $\widetilde{w}^{(0)} \leftarrow \varnothing$; $\phi^{(0)} \leftarrow \varnothing$; $t \leftarrow 0$; $D^{(0)} \leftarrow D$;
$\theta^{(0)} \leftarrow$ initialize parameters of M; $w^{(0)} \leftarrow \{1, 1, \ldots, 1\}$;
$\theta^{(0)} \leftarrow \mathrm{TrainFewEpochs}(\theta^{(0)}, D^{(0)})$;
while M not converged and $t < T_{\max}$ **do**
  $(S^{(t)}_{\mathrm{label}}, S^{(t)}_{\mathrm{feature}}, S^{(t)}_{\mathrm{mix}}, \mathcal{R}^{(t)}) \leftarrow \mathrm{ErrorTyping}(D^{(t)}, \{\theta_k\}_{k \in W_r})$;
  // $\mathcal{R}^{(t)} = (\{Y^{(t)}_{i,\mathrm{cand}}\}, \{\mu^{(t)}_{i,c}\}, \{J^{(t)}_i\}, \{\bar{q}^{(t)}_i\}, \{\Delta^{(t)}_i\})$
  $(D^{(t+1)}, w^{(t+1)}) \leftarrow \mathrm{ErrorHandle}(D^{(t)}, w^{(t)}, S^{(t)}_{\mathrm{label}}, S^{(t)}_{\mathrm{feature}}, S^{(t)}_{\mathrm{mix}}, \mathcal{R}^{(t)}, \theta^{(t)})$;
  $(C^{(t+1)}, \widetilde{w}^{(t+1)}, \phi^{(t+1)}) \leftarrow \mathrm{CoreSelect}(D^{(t+1)}, w^{(t+1)}, C^{(t)}, \widetilde{w}^{(t)}, \phi^{(t)}, K, \theta^{(t)})$;
  $\theta^{(t+1)} \leftarrow \mathrm{TrainFewEpochs}(\theta^{(t)}, C^{(t+1)}, \widetilde{w}^{(t+1)})$;
  $M \leftarrow M(\theta^{(t+1)})$;
  $t \leftarrow t + 1$;
return $C^{(t)}, \widetilde{w}^{(t)}, M(\theta^{(t)})$;

D.1 Training-Signal-Driven Error Typing

Overall procedure and outputs. ErrorTyping corresponds to the training-signal-driven error typing module in the overall algorithm and runs before ErrorHandle and CoreSelect. Figure 3 illustrates the overall typing procedure, and Algorithm 2 presents its steps. The module first obtains $S^{(t)}_{\mathrm{risk}}$ through persistent high-risk screening (RiskScreen) and then identifies $S^{(t)}_{\mathrm{label}}$ through the label-consistency check (LabelCheck). Samples whose anomalies have not been explained by label errors remain in $S^{(t)}_{\mathrm{rest}}$ and undergo the feature-sensitivity check (FeatureCheck) to obtain $S^{(t)}_{\mathrm{feature}}$; the remaining samples are temporarily assigned to $S^{(t)}_{\mathrm{mix}}$. The module outputs these three error sets and the intermediate information $\mathcal{R}^{(t)}$ required for subsequent handling. The following explanations describe the calculations and decision rationales in this order.

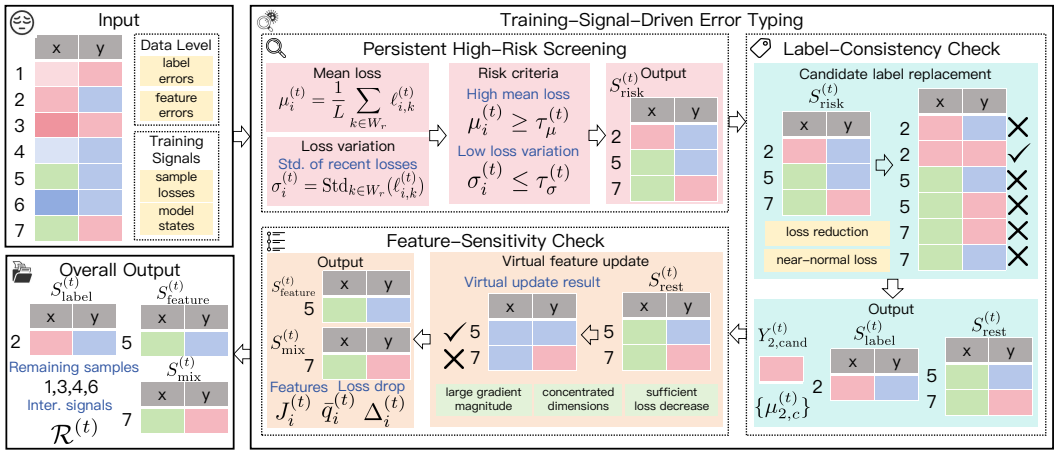


Figure 3: Training-signal-driven error typing framework.

(1) **Persistent High-Risk Screening.** To control loss-evaluation costs and reflect the model's recent training state, CleanCore retains parameter snapshots from the most recent L epochs up to the identification-triggering epoch r. Using losses $\ell^{(t)}_{i,k}$ within window W_r, the mean loss and loss variability are computed as $\mu^{(t)}_i = \frac{1}{L}\sum_{k \in W_r} \ell^{(t)}_{i,k}, \sigma^{(t)}_i = \sqrt{\frac{1}{L}\sum_{k \in W_r}\left(\ell^{(t)}_{i,k} - \mu^{(t)}_i\right)^2}$. Here, $\mu^{(t)}_i$ reflects the sample's overall recent loss level, while $\sigma^{(t)}_i$ reflects the stability of its loss changes. ErrorTyping considers both loss magnitude and variability across the most recent L consecutive

model snapshots, admitting only samples with high mean loss and low variability into the high-risk set. The mean-loss threshold controls the loss level required to enter this set, and the variability threshold controls the loss stability of high-loss samples. This condition screens for persistently and consistently high-loss samples, helping reduce the inclusion of hard but correct samples caused by transient high losses or training fluctuations.

(2) **Label-Consistency Check.** For each $z_i^{(t)} \in S_{\text{risk}}^{(t)}$, keep the input $x_i^{(t)}$ unchanged and virtually replace its label with $c \in Y$. The corresponding snapshot loss and window mean loss are

$$\ell_{i,k}^{(t)}(c) = \ell(f_{\theta_k}(x_i^{(t)}), c), \qquad k \in W_r,$$

$$\mu_{i,c}^{(t)} = \frac{1}{L} \sum_{k \in W_r} \ell_{i,k}^{(t)}(c).$$

If a label substantially reduces the mean loss relative to the original label and the replacement loss falls within an acceptable range, the sample is more likely to be a label-error sample. The candidate-label conditions in the main text therefore require both a loss reduction of at least $\tau_{\text{rel}}^{(t)}$ and a replacement mean loss no greater than $\tau_{\text{clean}}^{(t)}$.

The intuition is that other error types and the small number of remaining hard but correct samples typically have difficulty achieving a sufficient loss reduction toward the normal level through label replacement alone. This helps distinguish them from label-error samples. A sample is classified as a label error whenever at least one candidate label satisfies these conditions: $S_{\text{label}}^{(t)} = \left\{ z_i^{(t)} \in S_{\text{risk}}^{(t)} \mid Y_{i,\text{cand}}^{(t)} \neq \varnothing \right\}$. For every $c \in Y_{i,\text{cand}}^{(t)}$, ErrorTyping retains its mean loss $\mu_{i,c}^{(t)}$ as input to subsequent label handling. Compared with directly selecting the minimum-loss label, retaining the candidate set avoids premature hard relabeling and preserves uncertainty information for subsequent soft-label construction. Samples not classified as label errors form $S_{\text{rest}}^{(t)}$, where feature-gradient signals are then used to further distinguish feature errors from mixed errors.

(3) **Feature-Sensitivity Check.** Samples remaining after the label-consistency check may still contain feature errors, mixed errors, or anomalies that are difficult to classify at the current stage. ErrorTyping uses input-feature gradients to determine whether high loss is primarily driven by a few feature dimensions, then uses a lightweight virtual update to test whether adjusting these dimensions sufficiently reduces the loss.

For $z_i^{(t)} \in S_{\text{rest}}^{(t)}$, the feature vector is $x_i^{(t)} = (x_{i,1}^{(t)}, \ldots, x_{i,d}^{(t)})^\top$. Under snapshot $k \in W_r$, the input-feature gradient corresponding to the current label is

$$q_{i,k}^{(t)} = \nabla_{x_i^{(t)}} \ell(f_{\theta_k}(x_i^{(t)}), y_i^{(t)}) \in \mathbb{R}^d, \qquad q_{i,k}^{(t)} = (q_{i,k,1}^{(t)}, \ldots, q_{i,k,d}^{(t)})^\top.$$

Its window mean is

$$\bar{q}_i^{(t)} = \frac{1}{L} \sum_{k \in W_r} q_{i,k}^{(t)}, \qquad \bar{q}_i^{(t)} = (\bar{q}_{i,1}^{(t)}, \ldots, \bar{q}_{i,d}^{(t)})^\top.$$

First, denote the sum of absolute mean-gradient components by the total gradient magnitude: $A_i^{(t)} = \sum_{j=1}^d |\bar{q}_{i,j}^{(t)}|$. If $A_i^{(t)} < \tau_A^{(t)}$, the sample lacks a clear direction for feature correction in the sense of a first-order local correction and is no longer considered a feature-error candidate.

For samples satisfying the magnitude condition, sort $|\bar{q}_{i,j}^{(t)}|$ in descending order and let $J_i^{(t)}(m)$ denote the set of the top m feature indices. The cumulative gradient proportion over these features is

$$R_i^{(t)}(m) = \frac{\sum_{j \in J_i^{(t)}(m)} |\bar{q}_{i,j}^{(t)}|}{\sum_{j=1}^d |\bar{q}_{i,j}^{(t)}|}.$$

Given the maximum number m_0 of features that may be modified and the concentration threshold $\tau_{\text{conc}}^{(t)}$, select the smallest number of features satisfying the concentration requirement and determine the corresponding candidate feature set:

$$m_i^{(t)} = \min\{m \leq m_0 \mid R_i^{(t)}(m) \geq \tau_{\text{conc}}^{(t)}\}, \qquad J_i^{(t)} = J_i^{(t)}(m_i^{(t)}).$$

If such an $m_i^{(t)}$ exists, most of the sample's gradient is concentrated in no more than m_0 feature dimensions. In general, corrupted samples often have only a few anomalous features. When the gradient magnitude is large, this concentration suggests that high loss may be associated with local anomalies in a few features; ErrorTyping therefore uses it as a clue for identifying feature errors.

However, gradient concentration only indicates that the loss is sensitive to a few features and is insufficient by itself to establish a feature error. To further test whether local feature adjustments sufficiently reduce the loss under the current label, perform one lightweight virtual update on $J_i^{(t)}$ only for samples satisfying the magnitude and concentration conditions:

$$\widetilde{x}_{i,j}^{(t)} = \begin{cases} x_{i,j}^{(t)} - \eta_{\text{feat}} \operatorname{sign}(\bar{q}_{i,j}^{(t)}), & j \in J_i^{(t)}, \\ x_{i,j}^{(t)}, & j \notin J_i^{(t)}. \end{cases}$$

Here, $\eta_{\text{feat}} > 0$ is a small step size, and $\widetilde{x}_i^{(t)}$ is the virtually updated feature vector. This update is used only for a virtual test during identification and does not directly change feature values in the sample pool. Keeping $y_i^{(t)}$ unchanged, recompute the losses after the virtual update and their mean under the most recent L parameter snapshots:

$$\widetilde{\ell}_{i,k}^{(t)} = \ell(f_{\theta_k}(\widetilde{x}_i^{(t)}), y_i^{(t)}), \qquad \widetilde{\mu}_i^{(t)} = \frac{1}{L} \sum_{k \in W_r} \widetilde{\ell}_{i,k}^{(t)}.$$

The resulting mean loss reduction due to the virtual update is $\Delta_i^{(t)} = \mu_i^{(t)} - \widetilde{\mu}_i^{(t)}$. A sample is classified as a feature error only when its gradient magnitude is sufficiently large, its gradient is concentrated in a few feature dimensions, and the virtual update produces a sufficient loss reduction: $S_{\text{feature}}^{(t)} = \left\{ z_i^{(t)} \in S_{\text{rest}}^{(t)} \mid A_i^{(t)} \geq \tau_A^{(t)},\ m_i^{(t)} \text{ exists},\ \Delta_i^{(t)} \geq \tau_{\text{feat}}^{(t)} \right\}$. The high losses of the few remaining hard but correct samples are not caused by feature corruption. These joint conditions aim to reduce their misclassification as feature errors. ErrorTyping temporarily assigns those that fail these checks, together with other remaining samples that fail the conditions, to the mixed-error set: $S_{\text{mix}}^{(t)} = S_{\text{rest}}^{(t)} \setminus S_{\text{feature}}^{(t)}$. Samples in this set exhibit persistent high risk, and their anomalies are not sufficiently explained by label replacement alone or local feature adjustment alone under the current checks. They are therefore provisionally classified as suspected mixed errors for conservative handling. All three error sets represent current-stage decisions and may be further adjusted using new training signals in later iterations. ErrorTyping ultimately aims to improve downstream model performance and does not require every erroneous sample to be accurately and completely identified in a single iteration.

Algorithm 2: ErrorTyping

Input: $D^{(t)}$ (sample pool), cached training losses and model snapshots $\{\theta_k\}_{k \in W_r}$, label set Y, window W_r, threshold configuration $\boldsymbol{\tau}^{(t)}$, maximum feature number m_0, virtual step size η_{feat}

Output: $S_{\text{label}}^{(t)}, S_{\text{feature}}^{(t)}, S_{\text{mix}}^{(t)}, \mathcal{R}^{(t)}$

// Subroutines share the inputs above.

$S_{\text{risk}}^{(t)} \leftarrow \text{RiskScreen}(D^{(t)})$;

$\left(S_{\text{label}}^{(t)}, \{Y_{i,\text{cand}}^{(t)}\}, \{\mu_{i,c}^{(t)}\}\right) \leftarrow \text{LabelCheck}(S_{\text{risk}}^{(t)})$;

$S_{\text{rest}}^{(t)} \leftarrow S_{\text{risk}}^{(t)} \setminus S_{\text{label}}^{(t)}$;

$\left(S_{\text{feature}}^{(t)}, \{J_i^{(t)}\}, \{\bar{q}_i^{(t)}\}, \{\Delta_i^{(t)}\}\right) \leftarrow \text{FeatureCheck}(S_{\text{rest}}^{(t)})$;

$S_{\text{mix}}^{(t)} \leftarrow S_{\text{rest}}^{(t)} \setminus S_{\text{feature}}^{(t)}$;

$\mathcal{R}^{(t)} \leftarrow \left(\{Y_{i,\text{cand}}^{(t)}\}, \{\mu_{i,c}^{(t)}\}, \{J_i^{(t)}\}, \{\bar{q}_i^{(t)}\}, \{\Delta_i^{(t)}\}\right)$;

return $S_{\text{label}}^{(t)}, S_{\text{feature}}^{(t)}, S_{\text{mix}}^{(t)}, \mathcal{R}^{(t)}$;

(4) **Reliability and Complexity.** ErrorTyping provides the basis for subsequent error handling and coreset selection. Since CleanCore identifies errors iteratively, a single identification step need not cover every erroneous sample; greater attention is paid to whether samples assigned to each error set actually belong to the corresponding type. This subsection therefore analyzes the typing reliability of identified samples under the stated assumption and provides the corresponding computational complexity analysis.

Assumption 1 (Approximate consistency of training updates and correctness of ideal decisions)**.** *At the stage under analysis, consider the actual training process and an ideal clean training process starting from the same model at the beginning of the window. Within W_r, their parameter updates are approximately consistent in direction and magnitude. Let G_k be the actual parameter-update vector at epoch k and $G_{\mathrm{clean},k}$ the corresponding update in the ideal clean process. There exists a small constant ε such that*

$$\|G_k - G_{\mathrm{clean},k}\|_2 \le \varepsilon, \qquad k \in W_r.$$

Furthermore, for samples identified as erroneous at the current stage, the guiding signals produced by the ideal clean training trajectory yield correct judgments of their true types under the same decision rules and thresholds at that stage.

Here, $G_k = \theta_k - \theta_{k-1}$ represents the actual parameter change between consecutive snapshots. Since the trajectories share the same starting point and their update difference is at most ε per epoch, their parameter difference at any epoch within the window is at most $L\varepsilon$. Let $\widehat{s}_i$ denote the actual decision-signal vector, including recent loss statistics, candidate-label losses, and feature gradients. Let s_i^* be the corresponding signals for the same current sample under the ideal clean training trajectory. Combining this parameter-difference bound with the local Lipschitz property of the decision signals with respect to model parameters within the window gives

$$\|\widehat{s}_i - s_i^*\|_\infty \le C_s\varepsilon.$$

Here, $C_s > 0$ is a combined constant incorporating the window length and the local sensitivity of the signals. Under these conditions, the parameter-update deviation can thus be converted into an upper bound on the decision-signal deviation.

Let $\widehat{T}_i$ denote the sample type actually assigned at the current stage, and let T_i^* be the type obtained from s_i^* using the same decision rules and thresholds at that stage. The type space comprises clean, label, $\mathrm{feature}$, and mix; clean means that the sample is not identified as erroneous at the current stage. For samples identified as erroneous, when the preceding signal-deviation bound holds, we analyze the consistency between actual and ideal decisions through the margin between the ideal decision signal and its decision boundary. This boundary is not the model's classification boundary; it is the boundary of the error-type partition formed by the thresholds in ErrorTyping. Define the margin as

$$d_i = \mathrm{dist}_\infty(s_i^*, \partial R_{T_i^*}).$$

Here, dist_∞ is distance in the infinity norm, $R_{T_i^*}$ is the decision region for ideal type T_i^*, and $\partial R_{T_i^*}$ is its boundary. A larger d_i means that the ideal signal is farther from the boundary and its decision is less likely to change under small perturbations. For clarity and to emphasize the main analysis, stage superscripts are omitted from Theorem 1 onward in the theorems and their proofs; all variables denote the corresponding quantities at the stage under analysis.

Theorem 1 (Typing reliability of ErrorTyping)**.** *Under Assumption 1 and the preceding signal-deviation bound, ErrorTyping's current-stage probability P_{id} of inconsistent typing conditional on an error decision has a controllable upper bound.*

Proof. If $d_i > C_s\varepsilon$, the actual decision signal cannot cross the decision boundary, and therefore

$$d_i > C_s\varepsilon \quad \Longrightarrow \quad \widehat{T}_i = T_i^*.$$

Thus, for samples identified as erroneous, disagreement with the ideal type can occur only near the decision boundary. Hence,

$$P_{\mathrm{id}} = \Pr(\widehat{T}_i \ne T_i^* \mid \widehat{T}_i \ne \mathrm{clean}) \le \Pr(d_i \le C_s\varepsilon \mid \widehat{T}_i \ne \mathrm{clean}).$$

The upper bound on P_{id} is jointly determined by the parameter-update deviation ε, the decision-signal perturbation constant C_s, and the proportion of samples near the boundary among those identified as erroneous. Under Assumption 1, a smaller ε narrows the boundary neighborhood in which decisions may change; when the proportion of samples falling within this neighborhood is low, P_{id} can remain low. □

The above analysis provides a conditional guarantee when Assumption 1 and the signal-deviation bound hold. The ErrorTyping experiments in Appendix E.5 (1) provide further empirical support for its actual typing reliability.

For computational complexity, let c_ℓ and c_x denote the costs of one loss calculation and one input-gradient calculation for a sample, respectively. When identification is triggered, ErrorTyping evaluates the current pool using the most recent L snapshots. It reuses the losses of samples that participated in training during the previous stage under those snapshots, while recomputing and temporarily caching the other required losses. Since L is a preset fixed window length, only one mean and standard-deviation calculation over this loss window is needed per sample; loss evaluation and statistics therefore cost $O(n^{(t)}c_\ell)$. Likewise absorbing the fixed factor L in the other window computations into constants, the main cost is bounded by

$$T_{\mathrm{id}} = O\Big(n^{(t)}c_\ell + |S^{(t)}_{\mathrm{risk}}|(c_\ell + |Y|) + |S^{(t)}_{\mathrm{rest}}|(c_x + c_\ell + d\log d)\Big).$$

The first term accounts for loss evaluation and mean and standard-deviation statistics over the current pool. The second accounts for candidate-label checking on high-risk samples: under each snapshot, candidate labels share the outputs from a single forward pass, after which their losses are calculated and compared separately. The third accounts for input-gradient calculation, feature sorting, and loss evaluation after virtual updates on the remaining high-risk samples. Here, d is the feature dimension, and sorting costs $O(d\log d)$ per sample. Virtual-update evaluation is performed only for samples passing the magnitude and concentration checks; the size of the remaining set is used here to upper-bound that cost.

After successive screening, $S^{(t)}_{\mathrm{risk}}$ and $S^{(t)}_{\mathrm{rest}}$ usually contain substantially fewer than $n^{(t)}$ samples, keeping subsequent label and feature checks limited in scope. With fixed window length, class count, feature dimension, and model size, these main costs grow linearly with the sample size and are approximately $O(n^{(t)})$.

D.2 Error-Type-Guided Lightweight Error Handling

Overall Procedure and Weight Interpretation. This section corresponds to ErrorHandle in Algorithm 1. Between ErrorTyping and CoreSelect, this module performs lightweight processing of the sample pool according to the error sets and intermediate information from the preceding section and updates sample reliability weights, supplying the processed pool and weights for subsequent coreset selection. To uniformly represent different handling outcomes, CleanCore maintains a reliability weight $w_i^{(t)} \in [0,1]$ for each sample. It characterizes the trustworthiness of $z_i^{(t)}$ at training stage t and regulates its contribution to subsequent model updates, rather than precisely measuring its true data quality. Algorithm 3 presents the steps of ErrorHandle. Specifically, label-error softening (LES) updates the labels and weights of samples in $S^{(t)}_{\mathrm{label}}$; feature-error repair (FER) produces repaired feature representations and corresponding weights for samples in $S^{(t)}_{\mathrm{feature}}$; mixed-error samples are conservatively down-weighted (MDW); and samples not identified as erroneous at the current stage are gradually up-weighted (CWU). Finally, low-reliability samples are removed from the updated pool (LRS). The module outputs the processed pool $D^{(t+1)}$ and its reliability weights $w^{(t+1)}$.

(1) **Label-Error Softening (LES).** This step corresponds to LES in Algorithm 3. Step (2) of Section 3.1 identifies candidate label-error samples through the label-consistency check and retains their candidate label sets and corresponding mean losses. Soft labels are commonly used to handle label noise: existing methods typically assign probabilities to classes according to training losses to characterize label uncertainty. However, in iterative error correction, constructing soft labels directly over all classes may introduce interference from irrelevant classes and can be affected by fluctuations in detection at a single stage. This step therefore restricts the instantaneous soft-label distribution to the candidate set to reduce such interference, uses historical smoothing to mitigate detection fluctuations between stages, and adjusts sample weights according to label-correction confidence. These operations make the processed samples better suited to subsequent model updates and coreset selection. Figure 4 shows the overall framework.

Instantaneous Soft-Label Construction Based on the Candidate Set. For each $z_i^{(t)} \in S^{(t)}_{\mathrm{label}}$, step (2) of Section 3.1 provides the candidate set $Y^{(t)}_{i,\mathrm{cand}}$ and the corresponding mean losses $\mu^{(t)}_{i,c}$. ErrorHandle assigns higher probabilities to lower-loss candidates, obtaining the instantaneous soft-

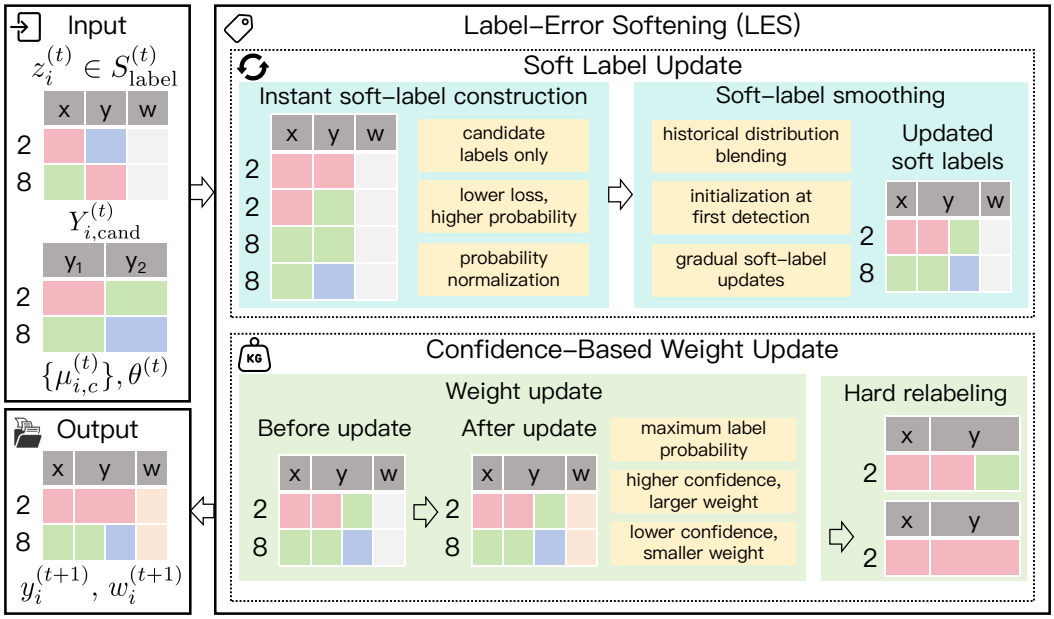


Figure 4: Label-error softening framework.

label distribution at the current stage:

$$\widehat{p}_i^{(t)}(c) = \frac{\exp(-\alpha \mu_{i,c}^{(t)})}{\sum_{c' \in Y_{i,\text{cand}}^{(t)}} \exp(-\alpha \mu_{i,c'}^{(t)})}, \qquad c \in Y_{i,\text{cand}}^{(t)},$$

$$\widehat{p}_i^{(t)}(c) = 0, \qquad c \notin Y_{i,\text{cand}}^{(t)}.$$

Here, $\widehat{p}_i^{(t)}(c)$ is the instantaneous judgment based only on the current candidate-label losses, and $\alpha > 0$ is a temperature coefficient controlling distribution concentration. Compared with direct hard relabeling, when multiple candidate labels exist, this distribution preserves uncertainty among them and provides input for historical smoothing, helping prevent a single detection result from being fixed prematurely.

Soft-Label Smoothing as Training Evolves. Since error detection is triggered periodically during training, loss fluctuations at a single stage may cause the candidate-label distribution to oscillate. For greater stability, ErrorHandle exponentially smooths the current instantaneous distribution $\widehat{p}_i^{(t)}(c)$ with the historical soft label $p_i^{(t)}(c)$. Both the first and subsequent updates follow

$$p_i^{(t+1)}(c) = (1-\beta) p_i^{(t)}(c) + \beta \widehat{p}_i^{(t)}(c), \qquad c \in Y.$$

Here, $0 < \beta < 1$ controls the mixing ratio between current detection and historical judgment. When a sample is first classified as a label error, the historical distribution is initialized with the one-hot distribution corresponding to its original label. The trainable soft label after the stage update is thus $y_i^{(t+1)} = \left(p_i^{(t+1)}(c)\right)_{c \in Y}$. During subsequent training and ErrorTyping, a soft-label sample's loss is computed as the probability-weighted sum of class-specific losses:

$$\ell_{i,k}\left(y_i^{(t+1)}\right) = \sum_{c \in Y} p_i^{(t+1)}(c) \ell_{i,k}^{(t)}(c).$$

Class-specific losses use the model parameters and sample features at the corresponding training or evaluation step, while the probability weights use the soft-label distribution maintained at the corresponding stage. A hard label can be viewed as a one-hot distribution, so the preceding loss notation applies uniformly to hard- and soft-label samples. Smoothing preserves continuity in historical judgments while allowing new training signals to gradually correct the distribution, reducing frequent switching among candidate classes.

Confidence-Based Lightweight Weight Updating. The concentration of the soft-label distribution can reflect confidence in label correction. ErrorHandle therefore characterizes this confidence by the dominant-label probability: $r_i^{(t+1)} = \max_{c \in Y} p_i^{(t+1)}(c)$. Using this confidence, the sample weight is updated linearly:

$$w_i^{(t+1)} = 1 - \lambda \left(1 - r_i^{(t+1)}\right), \qquad 0 < \lambda < 1.$$

Here, λ is the weight-adjustment coefficient. When $r_i^{(t+1)}$ approaches 1, the sample weight approaches 1. A lower $r_i^{(t+1)}$ indicates greater remaining label uncertainty, so a smaller weight is assigned to reduce the sample's influence on model updates.

When the same dominant class maintains high confidence over multiple consecutive stages, ErrorHandle may convert the soft label to that dominant class, $\arg\max_{c \in Y} p_i^{(t+1)}(c)$. Samples with insufficient confidence retain their soft labels, with lower weights limiting their training influence. This avoids premature hard relabeling while reducing interference from uncertain labels in model updates and subsequent coreset selection.

(2) **Feature-Error Repair (FER).** This step corresponds to FER in Algorithm 3 and describes lightweight iterative repair over candidate features for the feature-error samples identified in step (3) of Section 3.1. For each $z_i^{(t)} \in S_{\text{feature}}^{(t)}$, that step provides three quantities directly related to repair: the candidate feature set $J_i^{(t)}$, the mean feature gradient $\bar{q}_i^{(t)}$, and the virtual-update loss reduction $\Delta_i^{(t)}$. ErrorHandle uses these signals to determine repair dimensions and sample-specific step sizes, performs a few controlled updates on $J_i^{(t)}$ using instantaneous input gradients, and adjusts sample reliability weights according to actual repair gains. All feature types are handled in the model's input space: continuous features use numerical representations, integer features use continuous relaxations, and discrete categorical features are first one-hot encoded and then continuously relaxed during repair. Figure 5 shows the framework.

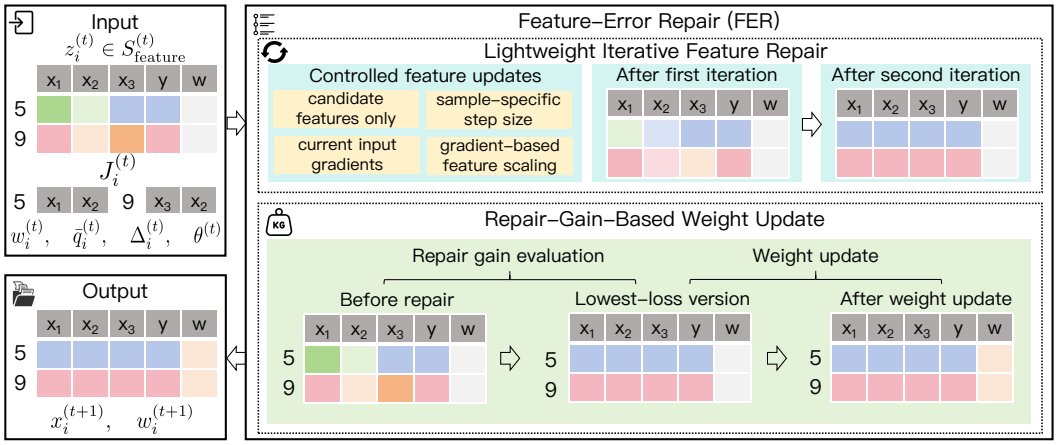


Figure 5: Feature-error repair framework.

Lightweight Iterative Repair on Candidate Features. At stage t, ErrorHandle treats sample features as variables to be repaired, keeps the label $y_i^{(t)}$ unchanged, and fixes the model parameters at $\theta^{(t)}$, the parameters when identification was triggered. Two iteration indices must be distinguished: t is the outer training stage, whereas u is a local step within feature repair.

Let $x_i^{(t,u)}$ denote the feature vector at repair step u. Since $u = 0$ denotes the original features before repair, initialize $x_i^{(t,0)} = x_i^{(t)}$. At step u, the model parameters are not updated; the current sample loss is computed only in the input-feature space: $\ell_i^{(t,u)} = \ell\left(f_{\theta^{(t)}}\left(x_i^{(t,u)}\right), y_i^{(t)}\right)$. Differentiating with respect to the current input features gives $q_i^{(t,u)} = \nabla_{x_i^{(t,u)}} \ell_i^{(t,u)}$. This instantaneous input gradient at repair step u differs from the window gradient $q_{i,k}^{(t)}$ in step (3) of Section 3.1. The latter

supports statistical decisions during detection, whereas the former supports local updates during actual repair. Both indicate the loss-sensitive directions of input features, but they serve different stages.

Since step (3) of Section 3.1 has already determined $J_i^{(t)}$, repair need not update every feature. Define the truncated gradient $\widetilde{q}_i^{(t,u)}$ by retaining only components on the candidate set:

$$\widetilde{q}_{i,j}^{(t,u)} = \begin{cases} q_{i,j}^{(t,u)}, & j \in J_i^{(t)}, \\ 0, & j \notin J_i^{(t)}. \end{cases}$$

Subsequent updates therefore affect only candidate features, leaving all others unchanged. This restricts repair to the few candidate dimensions screened in step (3) of Section 3.1 and avoids unnecessary perturbations to the full feature space.

Two update quantities must then be determined: a sample-specific step size controlling the overall repair magnitude, and feature-specific relative update coefficients controlling repair strength among candidate features. ErrorHandle normalizes the relative magnitudes of the current truncated gradients to obtain the latter:

$$a_{i,j}^{(t,u)} = \frac{|\widetilde{q}_{i,j}^{(t,u)}|}{\max_{h \in J_i^{(t)}} |\widetilde{q}_{i,h}^{(t,u)}| + \epsilon}, \qquad j \in J_i^{(t)}.$$

Here, ϵ is a tiny constant preventing a zero denominator. The candidate set $J_i^{(t)}$ is constructed from cumulative gradient proportions to cover a few dimensions explaining the main loss changes; membership does not imply equal error likelihood. In some cases, a few features contribute most of the gradient magnitude, but several small-gradient dimensions are still included to satisfy the concentration threshold. In addition to the gradient update itself, ErrorHandle therefore constructs $a_{i,j}^{(t,u)}$ from relative gradient magnitudes to further reduce updates to low-gradient dimensions. Repair is thus concentrated on candidates to which the current loss is more sensitive, reducing unnecessary perturbations to low-contribution dimensions.

The sample-specific step size controls the overall repair magnitude according to a first-order approximation: a larger loss reduction from the virtual update suggests greater repairability and permits a larger step. To control the magnitude, ErrorHandle normalizes the step by the mean-gradient scale from detection. The step size can therefore be set as

$$\eta_i^{(t)} = \eta_{\text{base}} \cdot \min\left(\eta_{\max}, \frac{\lambda_{\text{tar}} \Delta_i^{(t)}}{\sum_{j \in J_i^{(t)}} |\bar{q}_{i,j}^{(t)}|^2 + \epsilon}\right).$$

Here, η_{base} is a base scaling coefficient, $\eta_{\max}$ controls the upper limit, and λ_{tar} controls the desired repair magnitude. This expression connects the actual repair magnitude to the virtual repair effect observed during detection, constraining repair by current-stage repairability signals instead of blindly following gradient descent.

The feature update at step u is finally

$$x_{i,j}^{(t,u+1)} = x_{i,j}^{(t,u)} - \eta_i^{(t)} a_{i,j}^{(t,u)} \widetilde{q}_{i,j}^{(t,u)}.$$

Because $\widetilde{q}_i^{(t,u)}$ is zero outside the candidate set, this update changes only candidate dimensions. Features with lower and upper bounds are clipped to their valid ranges after updating to avoid unreasonable values. Repaired integer and categorical features retain their continuous relaxations for subsequent feature processing and model training. The repair aims to improve downstream training and does not require rediscretization into integer values or a single category.

Repair uses a few controlled iterations. After every update, ErrorHandle recomputes the feature loss and records the lowest-loss version. Iteration stops when successive updates offer limited gains or the step count reaches M_{feat}. The lowest-loss version is selected as the candidate repair. The initial state $u = 0$ is also included in this comparison: if repair yields no effective improvement, the original features are retained, preventing unreliable modifications from entering the pool.

Repair-Gain-Driven Weight Maintenance. After feature repair, ErrorHandle updates the sample weight according to the final outcome. Let $u_i^\star$ be the best repair step selected upon stopping, with features $x_i^{(t,u_i^\star)}$. The features written to the next-stage pool are $x_i^{(t+1)} = x_i^{(t,u_i^\star)}$. To measure repair effectiveness, define the gain as the loss reduction before and after repair: $\Delta_{i,\text{fix}}^{(t+1)} = \ell_i^{(t,0)} - \ell_i^{(t,u_i^\star)}$. A large gain indicates that adjusting a few candidate features substantially reduces the loss, making a repairable feature error more likely. A small or zero gain indicates that limited-step repair under the current model has not substantially improved the loss, leaving its effectiveness uncertain. To feed the outcome into subsequent training, ErrorHandle normalizes the gain into repair confidence:

$$\rho_i^{(t+1)} = \left[\frac{\Delta_{i,\text{fix}}^{(t+1)}}{\ell_i^{(t,0)} + \epsilon} \right]_0^1.$$

Here, $[\cdot]_0^1$ denotes clipping to $[0, 1]$. The sample weight is then smoothly updated toward this confidence:

$$w_i^{(t+1)} = (1 - \eta_w) w_i^{(t)} + \eta_w \rho_i^{(t+1)}.$$

Here, $0 < \eta_w < 1$ is the weight-smoothing coefficient. Greater repair gains produce higher $\rho_i^{(t+1)}$ and higher sample weights, allowing repaired samples to participate more fully in subsequent training. Limited gains produce lower confidence and weights, weakening the sample's influence on model updates and coreset selection.

(3) **Reliability Maintenance for Other Samples.** The mixed-error set $S_{\text{mix}}^{(t)}$ comprises the remaining high-risk samples whose anomalies cannot be sufficiently explained by label replacement or repair of a few features in step (3) of Section 3.1. Since direct relabeling, repair, or removal could amplify misclassification, ErrorHandle conservatively down-weights these samples (MDW), first reducing their training contribution and continuing to observe retained samples in subsequent stages.

Given the current reliability weight $w_i^{(t)}$ of $z_i^{(t)}$, a mixed-error decision triggers the multiplicative update

$$w_i^{(t+1)} = \lambda_{\text{mix}} w_i^{(t)}, \qquad 0 < \lambda_{\text{mix}} < 1.$$

Here, λ_{mix} controls the down-weighting of mixed-error samples to weaken their current training contribution. For samples not identified as erroneous at this stage, ErrorHandle gradually restores their contribution through slow up-weighting (CWU):

$$w_i^{(t+1)} = \min\{w_i^{(t)} + \eta_{\text{clean}}, 1\}.$$

Here, η_{clean} is a small up-weighting step. This strengthens supervision from relatively reliable samples while avoiding excessive weight increases based on a single-stage judgment.

Following the typing intuition in Section 3.1, we expect that even if a few difficult but correct samples enter $S_{\text{risk}}^{(t)}$, they are likely to be temporarily assigned to $S_{\text{mix}}^{(t)}$. ErrorHandle leaves their features and labels unchanged and initially only down-weights them. If they are retained and are no longer identified as erroneous in later stages, their weights can gradually recover.

After updating all sample weights, low-reliability sample removal (LRS) checks only samples in the current high-risk set $S_{\text{risk}}^{(t)}$. Those with updated weights below the removal threshold $w_{\min}$ are removed from $D^{(t+1)}$ and no longer participate in coreset selection, preventing persistently unreliable samples from consuming the training budget.

(4) **Reliability and Complexity.** Building on reliable ErrorTyping decisions, this subsection analyzes ErrorHandle's control of perturbations from identified erroneous samples. ErrorHandle does not restore erroneous samples to their ideal clean states in a single step; it corrects them progressively during training. We therefore analyze how label softening, feature repair, and reliability weighting jointly control residual gradient deviations and provide the corresponding upper bound and computational complexity.

Theorem 2 (Controllability of error-handling bias)**.** *With model parameters fixed and the joint contraction conditions below satisfied, the weighted total gradient deviation E_{handle} after ErrorHandle has a controllable upper bound that contracts relative to its pre-handling counterpart.*

Algorithm 3: ErrorHandle

Input: $D^{(t)}$ (sample pool), $w^{(t)}$ (sample weights), $S^{(t)}_{\text{label}}$, $S^{(t)}_{\text{feature}}$, $S^{(t)}_{\text{mix}}$ (where $S^{(t)}_{\text{risk}} = S^{(t)}_{\text{label}} \cup S^{(t)}_{\text{feature}} \cup S^{(t)}_{\text{mix}}$), $\mathcal{R}^{(t)}$ (signals from *ErrorTyping*), $\theta^{(t)}$, $w_{\min}$ (removal threshold), other handling hyperparameters and maintained per-sample history

Output: $D^{(t+1)}$ (processed sample pool), $w^{(t+1)}$ (updated sample weights)

$(\{Y^{(t)}_{i,\text{cand}}\}, \{\mu^{(t)}_{i,c}\}, \{J^{(t)}_i\}, \{\bar{q}^{(t)}_i\}, \{\Delta^{(t)}_i\}) \leftarrow \mathcal{R}^{(t)}$;

$D^{(t+1)} \leftarrow D^{(t)}$, $w^{(t+1)} \leftarrow w^{(t)}$;

for $z^{(t)}_i \in S^{(t)}_{\text{label}}$ **do**

  $(y^{(t+1)}_i, w^{(t+1)}_i) \leftarrow \text{LES}(z^{(t)}_i, Y^{(t)}_{i,\text{cand}}, \{\mu^{(t)}_{i,c}\}, \theta^{(t)})$;

  `// LES: label-error softening`

for $z^{(t)}_i \in S^{(t)}_{\text{feature}}$ **do**

  $(x^{(t+1)}_i, w^{(t+1)}_i) \leftarrow \text{FER}(z^{(t)}_i, w^{(t)}_i, J^{(t)}_i, \bar{q}^{(t)}_i, \Delta^{(t)}_i, \theta^{(t)})$;

  `// FER: feature-error repair`

for $z^{(t)}_i \in S^{(t)}_{\text{mix}}$ **do**

  $w^{(t+1)}_i \leftarrow \text{MDW}(w^{(t)}_i)$;

  `// MDW: mixed-error down-weighting`

for $z^{(t)}_i \in D^{(t)} \setminus (S^{(t)}_{\text{label}} \cup S^{(t)}_{\text{feature}} \cup S^{(t)}_{\text{mix}})$ **do**

  $w^{(t+1)}_i \leftarrow \text{CWU}(w^{(t)}_i)$;

  `// CWU: clean weight update`

$(D^{(t+1)}, w^{(t+1)}) \leftarrow \text{LRS}(D^{(t+1)}, w^{(t+1)}, S^{(t)}_{\text{risk}}, w_{\min})$;

`// LRS: low-reliability sample removal`

return $D^{(t+1)}, w^{(t+1)}$;

Proof. All gradients below are gradients of an individual sample's loss with respect to model parameters, evaluated at the same fixed parameters. Each error set below contains only samples retained after handling at the current stage, with sample identities used to match their pre- and post-handling states.

Let g^0_i, $\widetilde{g}_i$, and g^*_i denote the sample's gradients before handling, after current-stage handling, and in its ideal clean state, respectively. Define the pre-handling residual as $r^0_i = \|g^0_i - g^*_i\|_2$.

Let w^0_i and w_i denote the nonnegative sample reliability weights before and after handling, respectively.

For label-error samples, under relatively reliable identification, candidate labels and their mean losses provide evidence for soft-label correction, making label updates more likely to approach the ideal labels. Reliability weights further regulate the processed samples' contributions to training. Under the joint contraction conditions stated in the theorem, there exists $0 < \alpha_{\text{label}} < 1$ such that samples in this set satisfy

$$\widetilde{r}_{i,\text{label}} = \|\widetilde{g}_i - g^*_i\|_2, \qquad w_i \widetilde{r}_{i,\text{label}} \le \alpha_{\text{label}} w^0_i r^0_i.$$

For feature-error samples, candidate features, mean feature gradients, and virtual repair gains provide evidence for repair locations and step sizes, making controlled repair more likely to move toward the clean state. Post-handling reliability weights further regulate their residual influence. Under the joint contraction conditions stated in the theorem, there exists $0 < \alpha_{\text{feature}} < 1$ such that samples in this set satisfy

$$\widetilde{r}_{i,\text{feature}} = \|\widetilde{g}_i - g^*_i\|_2, \qquad w_i \widetilde{r}_{i,\text{feature}} \le \alpha_{\text{feature}} w^0_i r^0_i.$$

For retained mixed-error samples, the uncertainty of direct repair is high, so ErrorHandle only reduces their reliability weights without changing features or labels. Thus, with model parameters fixed, their unweighted gradient residuals remain unchanged:

$$\widetilde{r}_{i,\text{mix}} = \|\widetilde{g}_i - g^*_i\|_2 = \|g^0_i - g^*_i\|_2 = r^0_i.$$

According to the weight-update rule for mixed-error samples,

$$w_i = \lambda_{\text{mix}} w^0_i, \qquad 0 < \lambda_{\text{mix}} < 1,$$

their weighted residual influence satisfies $w_i\widetilde{r}_{i,\mathrm{mix}} = \lambda_{\mathrm{mix}} w_i^0 r_i^0$.

Combining the weighted residual perturbations from the three types gives the overall residual bound

$$R = \sum_{z_i \in S_{\mathrm{label}}} w_i \widetilde{r}_{i,\mathrm{label}} + \sum_{z_i \in S_{\mathrm{feature}}} w_i \widetilde{r}_{i,\mathrm{feature}} + \sum_{z_i \in S_{\mathrm{mix}}} w_i \widetilde{r}_{i,\mathrm{mix}}.$$

Let

$$S = S_{\mathrm{label}} \cup S_{\mathrm{feature}} \cup S_{\mathrm{mix}}, \qquad R^0 = \sum_{z_i \in S} w_i^0 r_i^0,$$

where R^0 is the sum of pre-handling weighted residuals for the same retained samples. Write

$$\bar{\alpha} = \max\{\alpha_{\mathrm{label}}, \alpha_{\mathrm{feature}}, \lambda_{\mathrm{mix}}\} < 1.$$

The joint contraction conditions and the down-weighting rule for mixed-error samples give $R \leq \bar{\alpha} R^0$.

Let the weighted total gradient deviation of the identified erroneous samples retained after current-stage handling, relative to their ideal clean states, be

$$E_{\mathrm{handle}} = \left\| \sum_{z_i \in S} w_i (\widetilde{g}_i - g_i^*) \right\|_2.$$

The triangle inequality gives

$$E_{\mathrm{handle}} \leq \sum_{z_i \in S} w_i \|\widetilde{g}_i - g_i^*\|_2 = R \leq \bar{\alpha} R^0.$$

Thus, under the stated conditions, the weighted total gradient deviation after error handling is controlled by the contracted residual bound. When $R^0 \neq 0$, this bound is strictly smaller than the corresponding pre-handling residual sum. □

The ErrorHandle experiments in Appendix E.5 (2) fix model parameters and measure the proportion of samples whose weighted gradient residual decreases after handling. Across the first three executions on all five datasets, this proportion is at least 96.9%, showing that most evaluated samples exhibit a decrease in weighted residuals and providing sample-level empirical support for the joint suppression effect of sample correction and reliability weighting.

In terms of computational complexity, ErrorHandle primarily operates on identified candidate error sets, while slowly up-weighting samples not identified as erroneous, denoted by $S_{\mathrm{clean}}^{(t)}$. Label-error handling mainly reuses candidate-label losses from identification and constructs soft labels and updates weights. Feature-error handling performs at most M_{feat} controlled repair steps on at most m_0 candidate features. Mixed-error handling mainly down-weights samples. LRS checks the removal threshold only for $S_{\mathrm{risk}}^{(t)}$. Hence,

$$\begin{aligned} T_{\mathrm{handle}} = O\big(&|S_{\mathrm{label}}^{(t)}||Y_{i,\mathrm{cand}}^{(t)}| + |S_{\mathrm{feature}}^{(t)}| M_{\mathrm{feat}}(c_x + c_\ell + m_0) \\ &+ |S_{\mathrm{mix}}^{(t)}| + |S_{\mathrm{clean}}^{(t)}| + |S_{\mathrm{risk}}^{(t)}|\big). \end{aligned}$$

Operations on $S_{\mathrm{mix}}^{(t)}$, $S_{\mathrm{clean}}^{(t)}$, and $S_{\mathrm{risk}}^{(t)}$ are lightweight. Successive screening usually makes $S_{\mathrm{label}}^{(t)}$ and $S_{\mathrm{feature}}^{(t)}$ substantially smaller than the current pool, concentrating soft-label processing and feature repair on small candidate sets. The candidate-label set is bounded by the number of classes, $|Y|$, while repair dimensions and iterations are controlled by m_0 and M_{feat}, respectively. With fixed class count, feature dimension, model size, and control parameters, the per-sample costs have constant upper bounds, and every processed set is no larger than the current pool. The overall complexity of ErrorHandle is therefore bounded by $O(n^{(t)})$.

D.3 Reliability-Weighted Dynamic Coreset Selection

Overall Procedure and Design Rationale. This section corresponds to CoreSelect in Algorithm 1. Following ErrorHandle, this module constructs or updates the coreset for subsequent training based

on the processed sample pool $D^{(t+1)}$ and reliability weights $w^{(t+1)}$. Figure 6 shows the framework, and Algorithm 4 presents the steps. Specifically, CoreSelect first computes label-expected gradients (LabelExpectedGrad) and screens candidate representatives by reliability (ReliableCandidates). For an existing coreset, it evaluates representativeness through approximation-error evaluation (ApproxError). When no reusable coreset exists or its approximation error exceeds the rebuilding threshold, it performs reliability-weighted gradient coverage (RebuildCore) subject to budget K. Otherwise, it locally updates affected samples (LocalUpdate) and, if the error exceeds the local-update threshold, additionally maintains representative-cluster reliability (ClusterMaintain), preserving representativeness at a lower cost. Finally, CoreSelect aggregates sample reliability weights according to the resulting mapping $\phi^{(t+1)}$ (AggregateCoreWeights) to obtain coreset weights $\widetilde{w}^{(t+1)}$. Its outputs are $C^{(t+1)}$, $\widetilde{w}^{(t+1)}$, and $\phi^{(t+1)}$.

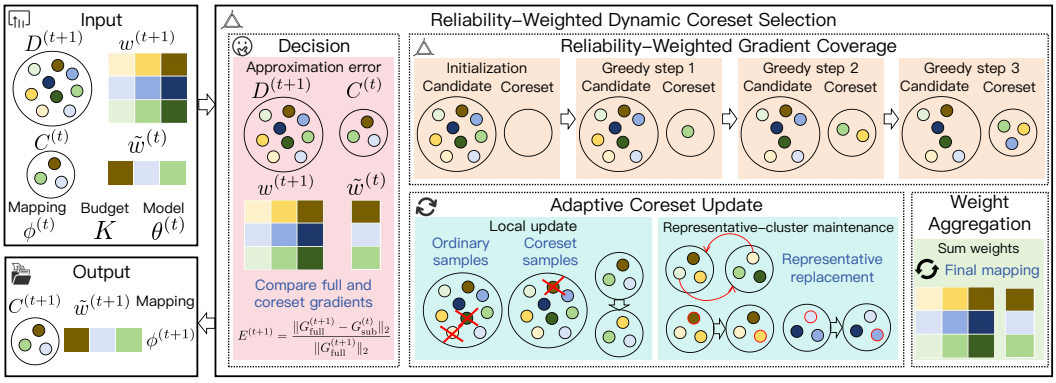


Figure 6: Reliability-weighted dynamic coreset selection framework.

(1) **Reliability-Weighted Gradient Coverage.** This step describes how to construct a coreset of at most K samples from $D^{(t+1)}$. Existing gradient-coverage methods indicate that when a coreset's weighted gradient approximates the full-data gradient, training on that coreset can obtain update directions close to full-data training. Unlike coreset selection that uses soft labels alone to characterize label uncertainty, CoreSelect further incorporates sample reliability weights after error handling, jointly considering the effects of label correction, feature repair, and mixed-risk suppression on training contributions.

Reliability-Weighted Gradient Coverage Objective. To unify hard- and soft-label samples, define the label-expected gradient of $z_i^{(t+1)}$ under parameters $\theta^{(t)}$ as

$$\bar{g}_i^{(t+1)} = \sum_{c \in Y} p_i^{(t+1)}(c) \nabla_\theta \ell \left(f_{\theta^{(t)}} \left(x_i^{(t+1)} \right), c \right).$$

A hard label is viewed as a one-hot distribution, while a soft label uses its probability distribution. Using $\bar{g}_i^{(t+1)}$ alone captures label uncertainty but cannot distinguish sample reliability after repair. CoreSelect therefore uses $w_i^{(t+1)}$ as a reliability coefficient and represents the full pool's reliability-weighted gradient as

$$G_{\text{full}}^{(t+1)} = \sum_{i=1}^{n^{(t+1)}} w_i^{(t+1)} \bar{g}_i^{(t+1)}.$$

For the constructed coreset, aggregating the updated sample weights according to the mapping $\phi^{(t+1)}$ yields the approximate gradient

$$G_{\text{sub}}^{(t+1)} = \sum_{i=1}^{n^{(t+1)}} w_i^{(t+1)} \bar{g}_{\phi^{(t+1)}(i)}^{(t+1)}.$$

The triangle inequality and nonnegative reliability weights give the approximation-error bound

$$\| G_{\text{full}}^{(t+1)} - G_{\text{sub}}^{(t+1)} \|_2 \leq \sum_{i=1}^{n^{(t+1)}} w_i^{(t+1)} \| \bar{g}_i^{(t+1)} - \bar{g}_{\phi^{(t+1)}(i)}^{(t+1)} \|_2.$$

This shows that the approximation error is controlled by the reliability-weighted gradient distances between all retained samples and their representatives. Compared with approximating label-expected gradients alone, this form pays greater attention to coverage errors for highly reliable samples and naturally weakens the influence of unreliable samples, remaining consistent with the preceding error-handling outcomes.

To reduce the risk of low-reliability samples being selected as representatives and repeatedly influencing subsequent coreset training, CoreSelect first constructs an initial candidate set using the reliability threshold $P^{(t+1)} = \{z_i^{(t+1)} \in D^{(t+1)} \mid w_i^{(t+1)} \geq \tau_{\text{rep}}\}$. Here, τ_{rep} is the representative-candidate reliability threshold, used only to screen representatives. If the initial candidate set contains fewer than K samples, samples from the remaining pool are added in descending order of reliability weight until the total candidate count reaches $\min(2K, n^{(t+1)})$; ties are resolved using a fixed sample order. The supplemented set is still denoted by $P^{(t+1)}$, and sample reliability weights remain unchanged. No supplementation is performed if the initial candidate count is at least K.

All samples in the processed pool still contribute to coverage-error calculation, but only members of the final candidate set $P^{(t+1)}$ may become representatives. Candidate-batch sampling can further reduce the search cost of greedy selection. The construction objective is therefore

$$C^{(t+1)} = \underset{C \subseteq P^{(t+1)},\, 1 \leq |C| \leq K}{\arg\min} \sum_{i=1}^{n^{(t+1)}} w_i^{(t+1)} \min_{z_j^{(t+1)} \in C} \|\bar{g}_i^{(t+1)} - \bar{g}_j^{(t+1)}\|_2.$$

Greedy Construction over Highly Reliable Candidates. The objective is a weighted gradient-coverage problem. Its coverage-gain function is monotone submodular; the representativeness guarantee for pointwise greedy selection is further analyzed in Theorem 3. CoreSelect introduces non-negative reliability weights and restricts the candidate space to $P^{(t+1)}$, focusing selection on highly reliable representatives. Each round therefore selects the candidate that maximally reduces coverage error. Specifically, for the current coreset C, define the reliability-weighted coverage error as

$$F(C) = \sum_{i=1}^{n^{(t+1)}} w_i^{(t+1)} \min_{z_j^{(t+1)} \in C} \|\bar{g}_i^{(t+1)} - \bar{g}_j^{(t+1)}\|_2.$$

The coreset is initialized with the sample giving the smallest weighted coverage error. Each subsequent round adds the sample from $P^{(t+1)} \setminus C$ that decreases $F(C)$ the most:

$$z_{j^*}^{(t+1)} = \underset{z_j^{(t+1)} \in P^{(t+1)} \setminus C}{\arg\max} \left[F(C) - F\left(C \cup \{z_j^{(t+1)}\} \right) \right].$$

This is equivalent to incrementally maximizing weighted gradient-coverage gain, selecting the candidate with the greatest coverage-error reduction each time. The formula specifies the pointwise greedy criterion. To reduce search costs, CoreSelect may also select approximate batches: each round samples and evaluates several candidate batches and adds the batch with the greatest error reduction.

After adding representatives, CoreSelect computes sample-to-new-representative gradient distances and compares them with the existing nearest distances. A closer distance immediately updates the mapping, avoiding another full mapping pass after construction. With batch selection, a tree index can be built over candidate representatives in each batch to quickly query the nearest candidate gradient distance and update mappings accordingly (Friedman et al., 1977).

This process repeats until budget K is reached or no candidates remain, producing $C^{(t+1)}$ and the mapping

$$\phi^{(t+1)}(i) = \underset{j:\, z_j^{(t+1)} \in C^{(t+1)}}{\arg\min} \|\bar{g}_i^{(t+1)} - \bar{g}_j^{(t+1)}\|_2.$$

Finally, the reliability weights of samples covered by the same representative are summed to obtain its coreset weight:

$$\widetilde{w}_j^{(t+1)} = \sum_{i:\phi^{(t+1)}(i)=j} w_i^{(t+1)}, \qquad z_j^{(t+1)} \in C^{(t+1)}.$$

Thus, $\widetilde{w}_j^{(t+1)}$ represents the aggregated reliable training contribution rather than a simple sample count. The resulting weighted coreset $(C^{(t+1)}, \widetilde{w}^{(t+1)})$ is the only data used for model updates in the subsequent stage.

(2) **Adaptive Coreset Update.** After error handling at stage t, label forms, feature representations, and reliability weights may all change. Repeating the greedy construction in step (1) of Section 3.3 at every stage would weaken the efficiency advantage of coreset training. CoreSelect therefore uses reliability-weighted hierarchical adaptive updating: when the current coreset remains representative, only local updates are performed; when representativeness decreases moderately, representative-cluster reliability is maintained in addition to local updates; and when the overall decline exceeds the rebuilding threshold, the coreset is reconstructed.

Global Validity Check. Continue to denote the preceding-stage coreset by $C^{(t)}$. Under the current model, compute the full weighted gradient $G_{\text{full}}^{(t+1)}$ using the processed sample pool $D^{(t+1)}$ and the updated sample reliability weights $w^{(t+1)}$; meanwhile, compute the approximate gradient $G_{\text{sub}}^{(t)}$ using the previous coreset $C^{(t)}$ and its preceding-stage aggregated weights $\widetilde{w}^{(t)}$. Here, the superscript t denotes the previous coreset and its weights; the gradient is still computed under the current model. The previous representatives and their original mapping are temporarily retained. Samples removed from the pool do not participate in computing $G_{\text{full}}^{(t+1)}$, but if they are representatives of the previous coreset, they are temporarily retained for evaluating $G_{\text{sub}}^{(t)}$ and are handled during subsequent rebuilding or local updating. Define the relative approximation error

$$E^{(t+1)} = \frac{\|G_{\text{full}}^{(t+1)} - G_{\text{sub}}^{(t)}\|_2}{\|G_{\text{full}}^{(t+1)}\|_2}.$$

When $\|G_{\text{full}}^{(t+1)}\|_2 = 0$, set $E^{(t+1)} = 0$ if $\|G_{\text{sub}}^{(t)}\|_2 = 0$; otherwise, set $E^{(t+1)} = +\infty$, triggering coreset rebuilding.

This error determines whether the current coreset still represents reliable training contributions after handling. If $E^{(t+1)} \le \tau_{\text{upd}}$, only affected samples are locally updated. If $\tau_{\text{upd}} < E^{(t+1)} \le \tau_{\text{reb}}$, representative-cluster reliability is additionally maintained. If $E^{(t+1)} > \tau_{\text{reb}}$, local repairs are insufficient to restore overall representativeness and coreset construction is repeated. Here, τ_{upd} and τ_{reb} are the local-update and rebuilding thresholds, respectively.

This trigger incorporates the combined effects of label correction, feature repair, and weight adjustment into the approximation error, making it more suitable for the training process with multiple dirty-data types considered here.

Local Updating of Affected Samples. If the global error permits local repair, CoreSelect updates only samples that may affect representative relationships instead of reconstructing the coreset.

Local updating first processes affected coreset representatives. A previous representative that has been removed or no longer satisfies the requirements of $P^{(t+1)}$ is removed from the representative set, and a highly reliable replacement is sought preferentially within its former coverage cluster. A representative that remains eligible is temporarily retained, with its updated label-expected gradient used in subsequent mapping updates. Processing the representatives yields the current effective coreset $C_{\text{eff}}^{(t+1)}$ and updated mappings.

Next, affected nonrepresentatives $z_i^{(t+1)} \notin C_{\text{eff}}^{(t+1)}$ still in the pool are processed. If the original representative has been removed or replaced, the sample is directly remapped. Otherwise, first compute its reliability-weighted deviation from its original representative:

$$d_i^{(t+1)} = w_i^{(t+1)} \|\bar{g}_i^{(t+1)} - \bar{g}_{\phi^{(t)}(i)}^{(t+1)}\|_2.$$

Here, $\phi^{(t)}$ is the preceding-stage mapping. If $d_i^{(t+1)} \le \tau_{\text{map}}$, the sample-mapping update threshold, the original mapping is retained and the new weight $w_i^{(t+1)}$ is used in aggregation. If $d_i^{(t+1)} > \tau_{\text{map}}$, the sample is mapped to the representative with the closest label-expected gradient in the effective coreset:

$$\phi^{(t+1)}(i) = \underset{j:\, z_j^{(t+1)} \in C_{\text{eff}}^{(t+1)}}{\arg\min} \|\bar{g}_i^{(t+1)} - \bar{g}_j^{(t+1)}\|_2.$$

Finally, after determining the effective representatives and mappings, CoreSelect reaggregates all coreset weights:

$$\widetilde{w}_j^{(t+1)} = \sum_{i:\phi^{(t+1)}(i)=j} w_i^{(t+1)}, \qquad z_j^{(t+1)} \in C_{\text{eff}}^{(t+1)}.$$

If representative-cluster reliability maintenance is also required, this aggregation takes place afterward.

Representative-Cluster Reliability Maintenance. When $\tau_{\text{upd}} < E^{(t+1)} \leq \tau_{\text{reb}}$, CoreSelect further checks the overall coverage quality of each representative cluster after sample-level local updates. Even an eligible representative may have a cluster dispersed by label updates, feature repair, or weight changes. For each representative $z_j^{(t+1)}$, define the within-cluster reliability-weighted mean deviation

$$D_j^{(t+1)} = \frac{\sum_{i:\phi^{(t+1)}(i)=j} w_i^{(t+1)} \|\bar{g}_i^{(t+1)} - \bar{g}_j^{(t+1)}\|_2}{\sum_{i:\phi^{(t+1)}(i)=j} w_i^{(t+1)}}.$$

If $D_j^{(t+1)} \leq \tau_{\text{clu}}$, the representative-cluster stability threshold, the cluster remains sufficiently stable in coverage and its representative is retained. If $D_j^{(t+1)} > \tau_{\text{clu}}$, the representative may remain reliable but can no longer compactly cover its assigned samples. A replacement minimizing within-cluster weighted gradient distances is therefore sought among eligible cluster members. After replacement, members of the original cluster are directly mapped to the new representative. This keeps the coreset aligned with the current pool's reliable training contributions at a low update cost.

Algorithm 4: CoreSelect

Input: $D^{(t+1)}$ (processed sample pool), $w^{(t+1)}$ (sample weights), $C^{(t)}$ (previous coreset), $\widetilde{w}^{(t)}$ (previous coreset weights), $\phi^{(t)}$ (previous mapping), coreset budget K, model parameters $\theta^{(t)}$, other selection hyperparameters

Output: $C^{(t+1)}$ (coreset), $\widetilde{w}^{(t+1)}$ (coreset weights), $\phi^{(t+1)}$ (sample-to-coreset mapping)

for $z_i^{(t+1)} \in D^{(t+1)}$ **do**
  $\bar{g}_i^{(t+1)} \leftarrow \text{LabelExpectedGrad}(z_i^{(t+1)}, \theta^{(t)})$;
$P^{(t+1)} \leftarrow \text{ReliableCandidates}(D^{(t+1)}, w^{(t+1)}, K)$;
if $C^{(t)} = \varnothing$ **then**
  $(C^{(t+1)}, \phi^{(t+1)}) \leftarrow \text{RebuildCore}(D^{(t+1)}, w^{(t+1)}, P^{(t+1)}, K)$;
else
  $E^{(t+1)} \leftarrow \text{ApproxError}(D^{(t+1)}, w^{(t+1)}, C^{(t)}, \widetilde{w}^{(t)}, \phi^{(t)})$;
  if $E^{(t+1)} > \tau_{\text{reb}}$ **then**
    $(C^{(t+1)}, \phi^{(t+1)}) \leftarrow \text{RebuildCore}(D^{(t+1)}, w^{(t+1)}, P^{(t+1)}, K)$;
  else
    $(C^{(t+1)}, \phi^{(t+1)}) \leftarrow \text{LocalUpdate}(\text{current state})$;
    if $E^{(t+1)} > \tau_{\text{upd}}$ **then**
      $(C^{(t+1)}, \phi^{(t+1)}) \leftarrow \text{ClusterMaintain}(\text{current state})$;
$\widetilde{w}^{(t+1)} \leftarrow \text{AggregateCoreWeights}(C^{(t+1)}, \phi^{(t+1)}, w^{(t+1)})$;
return $C^{(t+1)}, \widetilde{w}^{(t+1)}, \phi^{(t+1)}$;

(3) **Reliability and Complexity.** After identification and handling, CleanCore obtains a processed pool and its reliability weights. Coreset selection aims to reduce the training sample count while representing the pool's weighted gradient direction as closely as possible. This subsection further analyzes CoreSelect's representativeness guarantee and computational complexity.

The following analysis fixes the current-stage model, weighted sample pool, and candidate set.

Theorem 3 (Coreset representativeness guarantee)**.** *The reliability-weighted coverage gain is monotone submodular, and pointwise greedy selection under a size budget achieves the standard approximation guarantee; the coreset approximation deviation under nearest-representative mapping has a controllable upper bound.*

Proof. During full coreset construction, each sample is mapped to its nearest representative by label-expected gradient distance. By the triangle inequality, the coreset approximation deviation satisfies

$$E_{\text{core}} = \|G_{\text{sub}} - G_{\text{full}}\|_2 \leq \sum_i w_i \min_{z_j \in C} \|\bar{g}_i - \bar{g}_j\|_2 = F(C).$$

Thus, reducing $F(C)$ controls the approximation deviation. Selecting the candidate that most reduces $F(C)$ at each round directly compresses the upper bound on E_{core}.

To establish the effectiveness of greedy selection, convert gradient distance into similarity: $s_{ij} = M - \|\bar{g}_i - \bar{g}_j\|_2$, where M is sufficiently large to ensure $s_{ij} \geq 0$.

Define the reliability-weighted coverage gain

$$H(C) = \sum_i w_i \max_{z_j \in C} s_{ij}, \qquad H(\varnothing) = 0.$$

For a nonempty coreset C, the definitions give $H(C) = M \sum_i w_i - F(C)$. Maximizing $H(C)$ is therefore equivalent to minimizing $F(C)$. Moreover, $H(C)$ is monotone submodular: adding a core sample cannot reduce existing coverage gains, and a candidate's marginal gain diminishes as the selected coreset grows. Under $|C| \leq K$, let C_g be the coreset obtained by pointwise selection of the sample with the largest marginal gain from the fixed candidate set P, and let C^* be the optimal coreset under the same candidate set and size budget. Then

$$H(C_g) \geq (1 - \frac{1}{e}) H(C^*).$$

Greedy selection thus achieves the standard approximation guarantee for reliability-weighted coverage gain (Nemhauser et al., 1978). Together with $E_{\text{core}} \leq F(C_g)$ and the fact that larger $H(C_g)$ means smaller $F(C_g)$, this keeps the approximation deviation controllable as coverage error decreases. □

The above approximation guarantee applies to pointwise greedy selection. In practice, CoreSelect approximately optimizes the same reliability-weighted coverage objective through candidate-batch sampling and batch addition to reduce search costs. Full construction uses nearest-representative mapping, so the resulting coreset still satisfies $E_{\text{core}} \leq F(C)$.

For adaptive updating, CoreSelect evaluates global representativeness through the relative approximation error of reliability-weighted gradients, performs local updates according to the error level, and maintains representative clusters when necessary. Through local mapping adjustments and representative-cluster maintenance, adaptive updating aims to reduce the current coreset's gradient approximation deviation and maintain representativeness at a lower cost. The post-update deviation can still be controlled by the sum of reliability-weighted gradient distances under the actual mapping.

For computational complexity, let c_g be the cost of one gradient-distance calculation. Existing per-sample parameter gradients consistent with the current model parameters and processed sample states are reused directly, and the rest are computed as needed. With fixed model size and class count, gradient calculation, candidate screening, and error evaluation add only $O(n^{(t+1)})$ preprocessing and can be absorbed into the linear terms below. Full construction mainly costs candidate-batch sampling, nearest-distance queries from samples to candidate representatives, and mapping updates. Specifically, each round samples q candidate blocks from candidates in $P^{(t+1)}$ that have not yet been selected, with each block containing at most B candidates and its size not exceeding the remaining coreset budget or the number of remaining candidates, and adds the block with the greatest coverage-error reduction. Filling budget K thus takes approximately $\lceil \frac{K}{B} \rceil$ rounds. For each block, tree-index construction costs $O(B \log B)$, nearest-distance queries from all samples cost $O(n^{(t+1)} \log B \cdot c_g)$, and mapping updates and unified weight aggregation cost $O(n^{(t+1)})$. Therefore,

$$T_{\text{build}} = O\left(q \left\lceil \frac{K}{B} \right\rceil \left(B \log B + n^{(t+1)} \log B \cdot c_g + n^{(t+1)}\right) + n^{(t+1)}\right).$$

For a given model, c_g is a fixed computational unit. The number q of sampled candidate batches is a preset constant that does not grow with sample size, is typically much smaller than the sample count,

and can be absorbed into constants. Since nearest-distance queries from all samples to candidate blocks dominate full construction, this simplifies approximately to

$$T_{\text{build}} = O\left(\frac{n^{(t+1)} K \log B}{B}\right).$$

Compared with the $O((n^{(t+1)})^2 K)$ coverage-gain evaluation cost of pointwise greedy selection over all candidates, candidate-batch sampling and tree-based mapping therefore reduce full construction to approximately $O(\frac{n^{(t+1)} K \log B}{B})$, substantially lowering its cost.

At stages without full rebuilding, CoreSelect only locally maintains affected sample mappings and representative-cluster reliability, avoiding repeated candidate-batch sampling and full coverage-gain evaluation. The stage-level maintenance cost thus satisfies

$$T_{\text{core}} \leq T_{\text{build}}.$$

Equality holds when full rebuilding is triggered. With adaptive updating alone, the actual cost is usually substantially lower than T_{build}. Adaptive updating therefore reduces the number of full rebuilds and the total selection cost. In practice, candidate sampling and update-triggering strategies can be adjusted to predictive-performance and runtime-efficiency requirements, balancing representativeness against construction and maintenance costs.

D.4 Overall Analysis and Adaptive Threshold Settings

(1) **Overall Reliability and Training Complexity.** Theorems 1–3 provide analytical support under their respective conditions from three perspectives: error identification, error handling, and coreset representativeness. Building on these results, this subsection explains how CleanCore connects the three modules through stage-wise iteration by examining their specific mechanisms.

In the initial stage, research on early learning and MisDetect's use of early losses provide a basis for distinguishing sample quality using early training signals (Arpit et al., 2017; Deng et al., 2024). ErrorTyping aggregates recent loss statistics through a sliding window and combines label-replacement and feature-gradient signals for identification, reducing the influence of single-epoch training fluctuations on its decisions. In later stages, error handling and reliability weighting help reduce interference from low-quality samples in parameter updates, while coreset selection further prioritizes coverage of highly trustworthy training contributions. These mechanisms provide a rationale for Assumption 1. Theorem 1 then bounds the probability of inconsistent typing among samples identified as erroneous under the corresponding conditions, and the experiments in Appendix E.5 (1) provide further empirical support for actual typing reliability.

For identified erroneous samples, ErrorHandle uses soft-label smoothing to mitigate fluctuations in label judgments, restricts the locations and number of steps of feature repair, and retains the original features when there is no effective loss improvement. For candidate mixed-error samples whose types remain unclear, it prioritizes down-weighting. These designs aim to control the effects of erroneous handling and jointly suppress residual gradient interference through sample correction and reliability weighting, providing a mechanism-based explanation for the joint contraction conditions in Theorem 2. Appendix E.5 (2) further shows that most evaluated samples exhibit a decrease in weighted gradient residuals after handling, supporting this joint effect.

The processed samples and their reliability weights are further used in CoreSelect's coverage evaluation, allowing coreset selection to consider both the trustworthiness of training contributions and representativeness. Theorem 3 provides a theoretical basis for pointwise greedy construction and the corresponding control of gradient approximation deviation. The candidate-batch selection used in practice approximately optimizes the same coverage objective, while local updates and representative-cluster maintenance adapt to changes in sample states at a lower cost. New signals generated by coreset training are then fed back to ErrorTyping and ErrorHandle, allowing sample judgments, handling strategies, and representative selection to be continually adjusted as training evolves. When training signals provide useful information for distinguishing sample quality, this feedback helps reduce persistent interference from unreliable samples. When subsequent handling remains effective, the weighted residual gradient influence is expected to decrease further, supporting the maintenance of coreset representativeness and predictive performance. The above discussion

explains the connections among the modules and their design rationale, while the overall effect is evaluated through the main and supplementary experiments.

The preceding sections analyze each module's costs. CleanCore's total overhead consists of periodic additional processing and coreset training. Identification, handling, and selection are triggered by stage after several training epochs rather than every epoch. Their additional cost is therefore spread across multiple epochs and is usually much lower than the sample-access costs saved by subsequent coreset training. After selection, CleanCore retains the current parameters and continues training only on the coreset of at most K samples and its weights, without reinitializing the model or training on all data at every stage. Let E_{tr} be the number of training epochs per stage and c_{tr} the cost of a model update for one sample. CleanCore reduces stage-level training complexity from $O(E_{\mathrm{tr}} n^{(t+1)} c_{\mathrm{tr}})$ to $O(E_{\mathrm{tr}} K c_{\mathrm{tr}})$. Since K is much smaller than the current pool, it substantially reduces the samples accessed in each training epoch, lowering multiepoch training costs and improving overall training efficiency.

(2) **Adaptive Threshold Settings.** This subsection uniformly explains threshold settings in CleanCore. According to their roles, thresholds are divided into decision thresholds and trigger thresholds. Decision thresholds determine sample states and screen candidates; trigger thresholds control the execution of operations in error handling and coreset maintenance. We describe the setting principles of both types and then explain the method's tolerance to deviations in threshold settings.

For decision thresholds, CleanCore determines criteria according to current sample signals and screening requirements. For signals whose scales change during training, such as losses and gradients, thresholds adapt to the current-stage signal distribution. For example, the mean-loss threshold changes with the pool's loss distribution to screen high-risk samples. For signals with defined ranges, such as concentration and reliability, thresholds are preset according to screening requirements and reused in later stages. For example, the representative-candidate reliability threshold is used for initial candidate screening and remains unchanged in subsequent stages; when fewer than K candidates are available, the candidate set is supplemented according to the rule in Appendix D.3.

For a signal s using distribution statistics and a larger-value acceptance condition, CleanCore uses

$$\tau_s = \max\{Q_{\mathrm{pct}}(s), \mu_s + \kappa\sigma_s\}.$$

Here, $Q_{\mathrm{pct}}(s)$ is the quantile at a preset level in the corresponding signal set, μ_s and σ_s are its mean and standard deviation, and κ controls the required deviation from the mean. This rule considers both relative rank and deviation from the mean to control candidate screening. Signals with smaller-value acceptance conditions use corresponding lower-quantile thresholds. When deciding whether a loss approaches normal levels, the lower-quantile region of the current pool's mean-loss distribution serves as the reference.

In use, decision thresholds may follow a conservative strategy. CleanCore is a multistage iterative method and does not require a single detection round to cover every potential error. Samples not detected at the current stage and still retained in the pool can be reassessed using new loss and gradient signals in later stages. Conservative thresholds thus help improve the reliability of candidate error sets while retaining opportunities for later reassessment.

CleanCore also tolerates threshold-setting deviations within a reasonable range. Even if relatively loose criteria admit samples with less apparent error signals into candidate handling sets, subsequent handling does not modify all candidates equally strongly. For label errors, soft-label distributions and sample weights depend on candidate-label loss differences and dominant-label confidence. A sample without clear mislabeling typically has lower candidate-label confidence and will not be hard-relabeled prematurely. For feature errors, actual repair magnitude depends jointly on input-gradient strength, candidate-feature contributions, and loss reduction. Samples near decision boundaries typically receive small repairs. Thus, adjusting detection thresholds within a certain range usually does not substantially affect stability. Moderate relaxation mainly expands the candidate set without directly causing strong corrections; moderate tightening may delay handling of some samples, but they can still be reassessed in later stages. With thresholds in a reasonable range, CleanCore generally does not depend on the precise value of a single threshold.

For trigger thresholds, CleanCore presets criteria according to predictive-performance and runtime-efficiency requirements. For example, the removal threshold determines whether current high-risk samples that remain unreliable after handling should be removed, while the coreset rebuilding

threshold determines whether reconstruction is necessary. The former uses a conservative low-weight cutoff, preserving later observation opportunities for samples still worth retaining. The latter reflects application requirements for representativeness and maintenance costs. Other triggers are likewise determined by the corresponding operation's handling requirements and cost objectives and reused across stages when application objectives remain unchanged. For coreset rebuilding, an emphasis on efficiency permits a moderately higher threshold, allowing some approximation error and reducing full rebuilds. An emphasis on predictive performance permits a moderately lower threshold, triggering rebuilding after smaller declines in representativeness and maintaining it more promptly at some additional cost.

These two types cover the thresholds used in identification, handling, and selection, providing unified criteria for sample decisions and operation triggering. The relevant quantile parameters and preset cutoffs can be configured according to decision requirements and the emphasis on predictive performance or runtime efficiency, leaving room for adjustment to different applications.

E Supplementary Materials for Experiments

This section supplements the experimental settings and result analysis, covering overall evaluation, hyperparameter analysis, ablation experiments, further validation of error identification, error handling, and threshold sensitivity, and a random sampling control experiment.

E.1 Supplementary Explanations for Experimental Design

(1) **Datasets.** Of the eleven datasets, WDBC (Street et al., 1993), Banknote (Lohweg, 2012), Seismic Bumps (Sikora & Wrobel, 2010), Pendigits (Alpaydin & Alimoglu, 1996), Magic (Bock, 2004), Sensorless (Bator, 2013), MiniBooNE (Roe, 2005), Skin (Bhatt & Dhall, 2009), Covertype (Blackard, 1998), and SUSY (Whiteson, 2014) come from the UCI Machine Learning Repository. We use them as clean references before artificial error injection, injecting errors only into training sets and leaving test sets unchanged. SmartFactory comes from the REIN benchmark (Abdelaal et al., 2023) and includes dirty and corresponding clean reference versions for validation in a real-world dirty-data setting.

These datasets cover medical diagnosis, banknote authentication, seismic event prediction in mines, handwritten digit recognition, particle physics event classification, industrial sensor state identification, motor drive state diagnosis, neutrino event classification, skin pixel classification, forest cover type identification, and high-energy physics signal identification. They allow evaluation of CleanCore's predictive performance and runtime efficiency across tabular classification tasks.

Regarding scale, WDBC, Banknote, Seismic Bumps, Pendigits, Magic, and SmartFactory provide small- and medium-scale validation, while Sensorless, MiniBooNE, Skin, Covertype, and SUSY extend evaluation to larger datasets. The tasks include both binary and multiclass classification, with Pendigits, Sensorless, and Covertype supporting multiclass performance evaluation. Most datasets contain continuous features; Pendigits, SmartFactory, and Skin contain integer-valued numerical features. Covertype contains both continuous and categorical features, and Seismic Bumps contains both integer-valued numerical and categorical features.

Class distributions range from relatively balanced to markedly imbalanced. We use the imbalance ratio (IR), the ratio of the largest to smallest class counts in the training set before artificial error injection; SmartFactory uses the training labels of its clean counterpart. Sensorless has equal class counts, Pendigits has similar class counts, and Seismic Bumps and Covertype have relatively high IRs. These datasets therefore validate CleanCore across different scales, feature types, and class distributions. Table 2 provides their statistics.

Table 2: Dataset statistics.

Dataset	#Instances	#Features	Feature Type	#Classes	IR
WDBC	569	30	Continuous	2	1.69
Banknote	1,372	4	Continuous	2	1.25
Seismic Bumps	2,584	18	Integer-valued + Categorical	2	14.20
Pendigits	10,992	16	Integer-valued	10	1.08
Magic	19,020	10	Continuous	2	1.84
SmartFactory	23,645	18	Integer-valued	2	3.17
Sensorless	58,509	48	Continuous	11	1.00
MiniBooNE	130,064	50	Continuous	2	2.56
Skin	245,057	3	Integer-valued	2	3.82
Covertype	581,012	54	Continuous + Categorical	7	103.16
SUSY	5,000,000	18	Continuous	2	1.19

(2) **Baseline Methods.** We select eleven representative baselines and divide them into four categories by their main strategies: data cleaning methods (Cleanlab, MisDetect, and CTRL), coreset selection methods (Herding, DeepFool, GradMatch, and CoreTab), hybrid methods (GoodCore and Deem), and sequential combinations (CoreTab–MisDetect (C-M) and MisDetect–CoreTab (M-C)). The proposed CleanCore is also included. The methods are described below.

Cleanlab (Northcutt et al., 2021): Estimates label reliability using confident learning and retrains after removing suspected mislabeled samples.

MisDetect (Deng et al., 2024): Iteratively detects and removes suspected mislabeled samples using early training losses.

CTRL (Yue & Jha, 2024): Identifies and removes suspected mislabeled samples by clustering training-loss curves, then trains on the remaining data.

Herding (Welling, 2009): Greedily selects a representative training coreset by matching feature means between the coreset and the full dataset.

DeepFool (Moosavi-Dezfooli et al., 2016; Ducoffe & Precioso, 2018): Uses a tabular subset-selection adaptation based on DeepFool perturbation scores, prioritizing samples close to decision boundaries.

GradMatch (Killamsetty et al., 2021): Periodically selects and updates coresets during training by matching their weighted gradients to the full-data gradient, approximating full-data model updates.

CoreTab (Hadar et al., 2024): Builds data maps from training dynamics on tabular data and uses them to select representative core samples.

GoodCore (Chai et al., 2023): Selects core samples for data with missing values using expected distances in the space of possible repairs.

Deem (Deng et al., 2025): For mislabeled data, constructs soft labels from losses to represent label uncertainty and combines them with gradient matching to select and update coresets during training, reducing training costs.

C-M: First uses CoreTab to select a coreset from the original samples, then applies MisDetect error identification to that coreset, evaluating the sequential strategy of "selection before cleaning."

M-C: First uses MisDetect to identify and remove suspected mislabeled samples, then uses CoreTab to select a coreset from the processed data, evaluating "cleaning before selection."

CleanCore: Identifies error types in stages using training signals, performs lightweight error handling and updates sample reliability, and supports subsequent training through reliability-weighted dynamic coreset selection, continuously updating reliability and the coreset as training progresses.

(3) **Experimental Settings.** All experiments use the same random seed. On the same dataset with the same downstream model, all methods share the data split, initial error-injection instance, and model initialization, and use the same preprocessing and common training configurations.

The downstream classifiers are MLP and tabular ResNet (Gorishniy et al., 2021). MLP uses ReLU (Glorot et al., 2011) and BatchNorm (Ioffe & Szegedy, 2015), while tabular ResNet uses ReLU and LayerNorm (Ba et al., 2016); both use dropout (Srivastava et al., 2014) of 0.2. The optimizer is AdamW (Loshchilov & Hutter, 2017), with a learning rate of 1×10^{-3} and weight decay of 1×10^{-4}. Batch sizes and maximum epoch counts are set according to data scale and kept identical across methods on the same dataset. Training stops at the epoch limit or when the training loss fails to improve sufficiently over several consecutive epochs. The main comparison uses both downstream models; all other analyses use MLP unless otherwise specified. Each analysis varies only the factor under study, keeping the remaining configuration of the same method unchanged; special settings are stated separately.

For data splitting, WDBC, Banknote, Seismic Bumps, Magic, SmartFactory, Sensorless, MiniBooNE, Skin, and Covertype are split by stratified sampling into approximately 80% training and 20% test data. Pendigits uses its official training and test sets. Following SUSY's official description, its last $500{,}000$ samples form the test set and the other 4.5 million form the training set.

For error settings, random or distribution-shift errors are injected only into the training sets of the ten clean reference datasets, leaving test sets unchanged. SmartFactory uses its dirty version for training and its clean version for testing. The artificial error proportion is the fraction of training samples that are corrupted. For random errors, labels and categorical features are randomly replaced with other categories, while numerical features receive random numerical perturbations. For distribution-shift errors, labels are replaced according to a fixed mapping, and features shift in preset directions and magnitudes. Both settings include label, feature, and mixed errors, with mixed errors perturbing both the label and features of the same sample.

Experiments are conducted under Ubuntu 22.04 using Python 3.10 and PyTorch 2.1.0 (Paszke et al., 2019). Data processing and model training run on a 32-vCPU Intel(R) Xeon(R) Gold 6459C processor. On the same dataset with the same downstream model, all methods use the same number of threads or the same thread-allocation rules, ensuring absolute fairness.

(4) **Evaluation Metrics.** To comprehensively evaluate the methods in dirty-data training, we use Accuracy, Macro-F1, and runtime.

Accuracy is defined as

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}(\widehat{y}_i = y_i),$$

and measures overall classification accuracy on the test set. Here, N is the number of test samples, $\widehat{y}_i$ and y_i are predicted and true labels, and $\mathbf{1}(\cdot)$ is the indicator function. Higher Accuracy means a greater proportion of correctly classified samples.

Macro-F1 is defined as

$$\text{Macro-F1} = \frac{1}{C}\sum_{c=1}^{C}\frac{2\text{precision}_c \cdot \text{recall}_c}{\text{precision}_c + \text{recall}_c},$$

and equally averages class-specific F1 scores to measure predictive performance across classes (Opitz & Burst, 2019). Here,

$$\text{precision}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c}, \qquad \text{recall}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c}.$$

C is the number of classes, and TP_c, FP_c, and FN_c are the true-positive, false-positive, and false-negative counts for class c. Higher Macro-F1 indicates better overall predictive performance across classes. For brevity, Macro-F1 is subsequently called F1 in the main text.

Runtime is defined as

$$T_{\text{run}} = t_{\text{end}} - t_{\text{start}},$$

and measures the time required to complete an entire run under a given training configuration. A smaller value indicates greater efficiency.

E.2 Supplementary Explanations for Overall Evaluation

(1) **Comparison under Fixed Noise.** This part supplements the experimental settings and result analysis in Section 4.2 (1). Experiments compare methods on eleven datasets using MLP and tabular

Table 3: Predictive performance and runtime with tabular ResNet. CleanCore is our proposed method. ACC and F1 are percentages; TIME is in seconds. Darker and lighter blue backgrounds indicate the best and second-best distinct values in each dataset column, respectively; best values are bold, second-best values are underlined, and ties share the same formatting.

ACC (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Cleanlab	98.26	99.64	92.07	95.71	82.44	76.13	92.87	89.35	99.07	79.93	80.11
MisDetect	98.26	99.64	92.07	95.71	84.28	76.15	93.82	89.09	99.80	81.36	80.04
CTRL	98.26	99.64	92.07	95.71	83.60	76.02	93.63	89.35	99.48	79.21	79.57
Herding	91.30	96.36	90.91	90.02	79.71	71.16	79.45	81.43	87.07	70.59	60.65
DeepFool	94.78	81.09	91.68	92.82	79.24	77.44	55.32	74.99	86.19	64.62	55.71
GradMatch	96.52	98.55	91.10	95.17	82.94	74.71	92.17	85.10	98.56	64.25	77.09
CoreTab	93.04	89.09	86.65	89.42	79.11	74.37	90.52	86.77	99.24	79.57	57.57
GoodCore	94.78	96.73	90.91	91.34	81.18	77.61	86.02	86.51	99.69	82.03	79.53
Deem	93.91	98.55	90.91	93.80	84.02	75.09	79.06	87.19	99.16	69.39	78.38
C-M	96.52	97.45	91.49	92.34	80.76	74.79	88.07	84.07	79.25	73.62	71.81
M-C	98.26	99.27	85.30	95.54	83.42	76.49	93.92	88.80	99.84	79.78	67.35
CleanCore	**99.13**	**100.00**	**93.23**	**96.34**	**84.97**	**77.73**	**94.65**	**89.85**	**99.86**	**82.36**	**80.12**

F1 (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Cleanlab	98.12	99.63	54.31	95.71	79.41	43.74	92.81	86.66	98.61	53.36	79.71
MisDetect	98.12	99.63	52.37	95.70	81.94	43.75	93.81	86.73	99.70	53.11	79.73
CTRL	98.12	99.63	50.26	95.68	81.23	43.19	93.58	86.73	99.21	66.71	79.18
Herding	90.94	96.30	53.26	90.02	77.45	54.58	79.32	78.11	75.22	45.89	60.64
DeepFool	94.43	79.20	47.83	92.81	73.80	55.87	52.67	56.07	71.06	33.83	55.08
GradMatch	96.21	98.52	49.75	95.22	78.80	49.75	92.12	81.34	97.82	42.00	76.81
CoreTab	92.57	89.00	50.40	89.45	76.82	54.77	90.46	83.15	98.86	56.78	43.45
GoodCore	94.48	96.68	49.66	91.36	78.21	54.61	85.71	82.49	99.53	70.08	79.06
Deem	93.40	98.53	49.66	93.84	80.71	53.43	78.69	83.23	98.74	52.10	78.31
C-M	96.21	97.42	51.94	92.32	78.23	55.03	87.99	79.62	44.21	28.12	68.84
M-C	98.12	99.26	47.31	95.54	81.62	51.54	93.88	85.57	99.76	63.97	62.86
CleanCore	**99.07**	**100.00**	**55.55**	**96.38**	**82.99**	**56.50**	**94.63**	**87.47**	**99.79**	**70.71**	**79.76**

TIME (s) ↓

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Cleanlab	5.05	14.00	16.50	44.83	53.16	64.49	148.74	214.00	344.18	715.14	5582.80
MisDetect	3.37	10.73	14.56	46.16	57.10	48.37	61.44	126.14	187.54	777.51	5667.74
CTRL	5.30	10.71	16.68	26.39	14.52	13.80	69.57	166.82	171.91	735.54	7997.95
Herding	1.51	1.84	2.60	5.98	7.84	7.07	9.47	22.09	30.74	175.27	578.42
DeepFool	1.69	2.23	4.07	6.62	14.97	14.56	32.34	58.87	65.57	371.54	941.72
GradMatch	1.51	1.66	1.81	4.26	6.87	6.11	8.65	19.16	26.39	78.66	460.17
CoreTab	1.51	2.01	2.75	5.56	8.30	7.16	12.61	17.98	45.01	107.76	474.07
GoodCore	1.53	1.94	2.69	5.96	8.32	15.06	18.84	34.21	48.75	185.63	706.41
Deem	1.54	1.45	2.33	4.79	6.63	6.19	**5.84**	17.47	**17.06**	90.56	1723.96
C-M	1.51	2.12	3.39	6.78	8.32	7.50	16.99	29.01	57.16	148.89	464.07
M-C	1.96	2.78	2.34	7.37	13.61	14.64	30.66	120.39	116.28	264.73	1016.15
CleanCore	**1.47**	**1.34**	**1.71**	**3.57**	**6.58**	**5.59**	7.72	**14.45**	24.06	**78.15**	**456.63**

ResNet. For the ten clean reference datasets, 20% random errors are injected into the training sets. SmartFactory uses its existing dirty training set and clean test set to evaluate real-world dirty-data performance.

For methods with coreset selection, the budget is 10% on WDBC, Banknote, Seismic Bumps, Pendigits, Magic, and SmartFactory; 5% on Sensorless, MiniBooNE, Skin, and Covertype; and 1% on SUSY. All selection methods use the same budget on a given dataset, and both downstream models use identical budget settings. Unless otherwise specified, subsequent experiments retain these dataset-specific budgets.

Table 1 in the main text and Table 3 here report ACC, F1, and total runtime under MLP and tabular ResNet, respectively. With MLP, CleanCore achieves the best ACC on ten datasets and the best F1 on nine. With tabular ResNet, its ACC and F1 are best on all eleven datasets. CleanCore therefore maintains good predictive performance under both downstream models examined. In particular, it achieves the highest ACC and F1 under both models on SmartFactory, demonstrating good performance in the real-world dirty-data setting. On Seismic Bumps, which combines integer-valued numerical and categorical features with a relatively high imbalance ratio, CleanCore likewise achieves

the best ACC and F1 under both models, further showing that it performs well when mixed feature types and marked class imbalance coexist.

Compared with GradMatch, which also uses gradients for selection, CleanCore achieves higher ACC and F1 on every dataset under both downstream models, supporting the effectiveness of considering data quality in gradient-based coreset selection. Herding, DeepFool, and CoreTab do not consider data errors and thus struggle to ensure representativeness on erroneous data. GoodCore addresses only missing values and Deem only label errors, limiting their coverage of coexisting error types. For sequential methods, C-M selects before cleaning, leaving selection susceptible to erroneous samples. M-C cleans before selection but lacks a feedback mechanism that continuously updates reliability and the coreset during subsequent training. Cleanlab, MisDetect, and CTRL achieve high predictive performance on some datasets but mainly handle suspected mislabeled samples. In contrast, staged handling and reliability-weighted dynamic coreset selection allow CleanCore to reduce error interference and maintain representativeness, improving predictive performance.

For runtime efficiency, CleanCore has the shortest runtime on eight datasets with MLP and nine with tabular ResNet. It accelerates all datasets under both downstream models relative to the three cleaning methods, achieving approximately 2–24$\times$ speedups. This mainly results from coreset training reducing the samples accessed in each model-update epoch, while candidate-batch sampling, tree-based mapping, and adaptive updating control construction and maintenance costs. Cleaning methods still clean and train at a scale close to the full dataset, and M-C detects and handles errors on all data before selection, incurring extra costs. Furthermore, error handling helps reduce ineffective updates caused by erroneous samples and promote faster convergence, so CleanCore can take less total time than other selection methods even with its additional processing.

Overall, CleanCore achieves approximately 2–24$\times$ speedups over the three cleaning methods, with predictive performance close to or better than theirs. It also obtains better predictive performance with runtime efficiency close to or better than coreset methods, combining the advantages of cleaning and selection to improve both performance and efficiency.

(2) **Comparison across Noise Intensities and Types.** This part supplements the experimental settings and result analysis in Section 4.2 (2), comparing changes in the predictive performance of gradient-based coreset methods under different error intensities and types. We compare GradMatch, Deem, and CleanCore, all of which use gradient information to construct or update training coresets. GradMatch has no error-handling mechanism, Deem mainly handles label errors, and CleanCore simultaneously detects and handles label, feature, and mixed errors. This experiment therefore also compares more directly the effects of different handling approaches on gradient-based coreset selection.

Experiments use WDBC, Pendigits, Magic, MiniBooNE, Skin, and Covertype, injecting random or distribution-shift errors into training sets at 6%, 12%, 18%, 24%, and 30%. Figure 7 shows results on WDBC, Pendigits, Magic, and MiniBooNE; panels (a) and (b) correspond to random and distribution-shift errors, respectively.

As Figure 7 shows, CleanCore achieves the best ACC and F1 at every error proportion on the four datasets under both error settings. As the proportion increases, GradMatch and Deem decline more noticeably on some datasets, while CleanCore maintains high predictive performance overall, demonstrating its performance advantage under different error conditions. Table 4 further reports results on the larger Skin and Covertype datasets.

On Skin, all three methods perform well, but CleanCore consistently achieves the best results across both error types and all proportions. Its advantage is more pronounced on the more challenging Covertype dataset. At 30% errors, it achieves 81.03% ACC and 67.42% F1 under random errors, and 81.36% ACC and 71.68% F1 under distribution-shift errors, all clearly above GradMatch and Deem.

Overall, CleanCore achieves the best ACC and F1 on six datasets across both error types and different proportions, showing that it reduces the negative effects of different errors on model training and maintains good predictive performance as error type and intensity change.

(3) **Time Composition Analysis.** Using the same datasets and error settings as Section 4.2 (1), we measure the cumulative runtime of ErrorTyping, ErrorHandle, and CoreSelect over the entire run and compare time composition under MLP and tabular ResNet. In Table 5, Modules denotes their

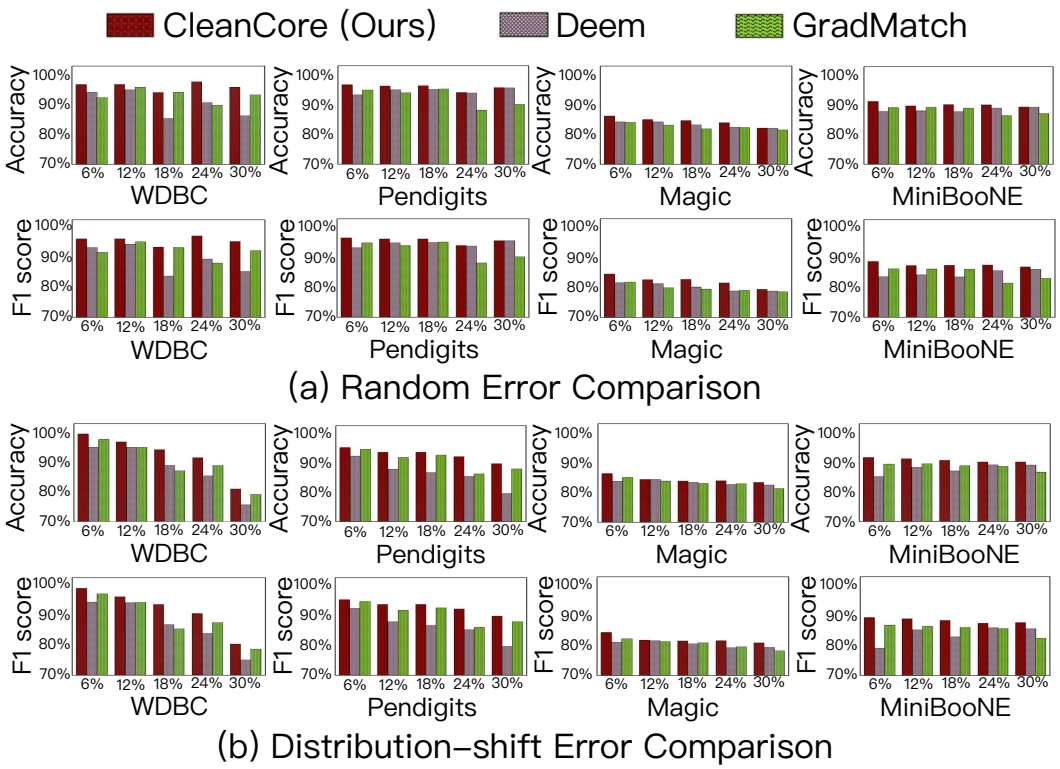


Figure 7: Predictive performance under different error types and proportions. The horizontal axis denotes the error proportion.

Table 4: Performance on Skin and Covertype under different error types and proportions (ACC/F1, %). Best results are bold; second-best results are underlined.

Noise Type	Dataset	Method	6%	12%	18%	24%	30%
Random	Skin	GradMatch	99.80/99.70	99.77/99.64	99.74/99.60	**99.79/99.68**	99.53/99.29
		Deem	99.40/99.10	99.50/99.24	99.41/99.11	99.22/98.83	99.27/98.90
		CleanCore	**99.81/99.71**	**99.78/99.67**	**99.77/99.66**	**99.79/99.68**	**99.76/99.64**
	Covertype	GradMatch	80.93/66.59	80.06/63.42	79.89/63.94	77.94/61.21	78.14/58.69
		Deem	70.24/48.07	73.64/52.31	73.91/51.42	74.42/51.20	74.27/49.06
		CleanCore	**84.55/76.00**	**83.74/74.55**	**82.44/70.89**	**81.82/68.87**	**81.03/67.42**
Shift	Skin	GradMatch	99.80/99.69	99.72/99.57	99.65/99.47	99.67/99.49	99.35/99.02
		Deem	99.53/99.29	99.19/98.78	99.12/98.68	99.22/98.83	99.04/98.56
		CleanCore	**99.82/99.73**	**99.75/99.62**	**99.75/99.63**	**99.73/99.59**	**99.55/99.32**
	Covertype	GradMatch	81.37/69.40	80.68/67.20	79.74/65.64	79.53/65.34	78.79/62.18
		Deem	65.67/49.22	67.26/49.91	68.39/50.28	67.82/46.57	68.02/46.37
		CleanCore	**84.78/77.20**	**83.92/74.96**	**82.87/72.58**	**82.41/72.32**	**81.36/71.68**

combined runtime, and Total denotes total runtime. $\frac{\text{Modules}}{\text{Total}}$ is the fraction of total runtime spent in the three modules, while $\frac{\text{CoreSelect}}{\text{Modules}}$ is the fraction of their combined runtime spent in CoreSelect.

Table 5 shows that the average module-time proportions under the two models are only 18.22% and 14.02%, respectively, even though coreset training has already reduced training costs. Relative to more expensive full-data training, the same module costs would account for an even smaller proportion. Within the modules, CoreSelect accounts for over 50% on ten datasets under each model. Although candidate-batch sampling, tree-based mapping, and adaptive updating already control its cost, CoreSelect remains the main source of module overhead, indicating that ErrorTyping and Er-

Table 5: Module-time proportions of CleanCore under two downstream models (%).

Dataset	MLP		ResNet	
	Modules / Total	CoreSelect / Modules	Modules / Total	CoreSelect / Modules
WDBC	6.51	65.38	5.32	35.21
Banknote	10.60	39.28	6.42	54.02
Seismic Bumps	10.25	82.07	10.42	56.69
Pendigits	20.77	62.22	21.69	84.79
Magic	18.96	67.85	10.36	82.10
SmartFactory	4.87	84.31	10.24	70.57
Sensorless	31.38	86.48	9.73	81.04
MiniBooNE	25.65	89.67	20.72	90.09
Skin	6.64	77.81	9.84	80.40
Covertype	44.20	95.76	22.38	94.98
SUSY	20.62	78.88	27.09	83.52

rorHandle together cost less in most settings. Lightweight CoreSelect, ErrorTyping, and ErrorHandle, together with fewer ineffective updates due to handling, allow CleanCore to outperform some coreset methods in runtime efficiency despite the additional modules.

E.3 Supplementary Explanations for Hyperparameter Analysis

All three experiments use Pendigits, Magic, Sensorless, MiniBooNE, and Covertype, with 20% random errors injected into training sets.

(1) **Coreset Budget.** This experiment analyzes the effects of coreset budget on CleanCore's predictive performance and efficiency. Given differences in dataset scale, the budgets for Pendigits and Magic are 4%, 8%, 12%, 16%, and 20%, while those for Sensorless, MiniBooNE, and Covertype are 2%, 4%, 6%, 8%, and 10%. Figure 8(a) shows Accuracy and runtime, with five consecutive bars per dataset representing its budgets in increasing order from left to right.

As Figure 8(a) shows, Accuracy generally increases with the budget on all five datasets. Increasing from the lowest to highest budget improves Accuracy by 0.88–4.33 percentage points, with more pronounced gains on Covertype and Sensorless, indicating that a larger coreset improves full-data coverage. Runtime also generally increases with budget, but overall efficiency remains high. In practice, an appropriate coreset size can be selected according to predictive-performance and efficiency requirements.

(2) **Sliding Window Length.** This experiment further analyzes the effects of sliding window length L on CleanCore's predictive performance and efficiency. The length is the number of model snapshots used to compute recent loss and gradient statistics during identification. We set L to 2, 4, 6, 8, and 10. Figure 8(b) shows Accuracy and runtime, with five consecutive bars per dataset representing these lengths in order from left to right.

As Figure 8(b) shows, Accuracy varies by only 0.10–0.83 percentage points across the five datasets, indicating insensitivity to window length. Lengths from $L = 4$ to $L = 6$ usually achieve high predictive performance because moderate windows balance the sufficiency and timeliness of training signals, more accurately characterizing the current model state. Runtimes are generally similar across window lengths and sometimes even shorter with a larger window. This indicates that window statistics account for a small and largely stable overhead; length has little influence on total runtime, which remains dominated by model training.

(3) **Number of Initial Training Epochs.** This experiment analyzes the effects of initial training epochs on the predictive performance and efficiency of gradient-based coreset methods. We select GradMatch, Deem, and CleanCore, all of which require initial training to obtain usable gradients

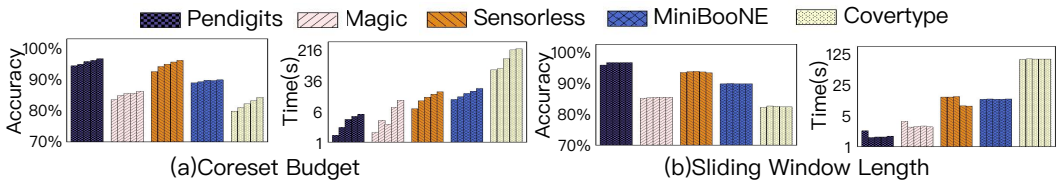


Figure 8: Sensitivity of CleanCore to coreset budget and sliding window length. Within each group, parameter values increase from left to right; their specific values are given in the text.

and training signals before first constructing a coreset. The initial epoch counts are 4, 8, 12, 16, and 20. Figure 9 shows the results.

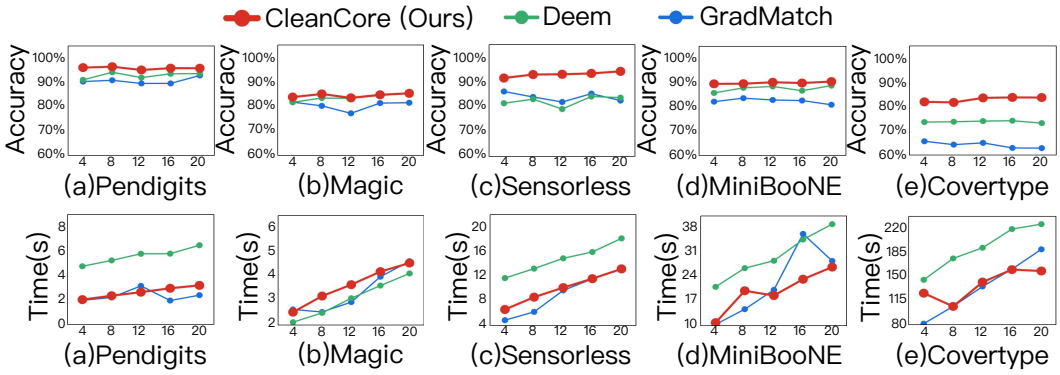


Figure 9: Accuracy and runtime under different initial training epoch counts. The horizontal axis denotes the number of initial training epochs.

For predictive performance, CleanCore achieves the highest Accuracy in all 25 settings formed by five datasets and five initial epoch counts. As the count changes, its Accuracy ranges span only 0.96–2.71 percentage points across datasets, showing overall stability. GradMatch and Deem fluctuate more noticeably on some datasets. This indicates that CleanCore is insensitive to the choice of initial epoch count and maintains high predictive performance across a broad range of values.

For runtime efficiency, the runtimes of all three methods generally increase with the initial epoch count. CleanCore's runtime costs are similar to GradMatch's and lower than Deem's in most settings. This is because its additional processing is inexpensive and helps reduce ineffective updates and promote convergence. On larger datasets such as MiniBooNE and Covertype, it maintains good efficiency under multiple initial epoch settings. Overall, **CleanCore maintains high predictive performance without requiring a precise initial epoch count while also retaining runtime efficiency.**

E.4 Ablation Experiments

To validate error handling, reliability weighting, feature repair, and dynamic coreset maintenance, we conduct ablations on Banknote, Seismic Bumps, Pendigits, Magic, SmartFactory, and Sensorless. The datasets cover mixed features and real-world dirty data for more comprehensive validation. SmartFactory uses its native dirty data; the other five training sets receive 40% random errors to create stronger error interference. The choice of this proportion follows the observation in MisDetect (Deng et al., 2024) that label-error proportions above 40% are rare in real applications.

No-Weight omits sample reliability weights to examine reliability weighting. No-Handle additionally removes ErrorTyping and ErrorHandle, selecting a coreset directly from the current pool for model training. CleanCore denotes the complete method with all modules. Figure 10 shows the results.

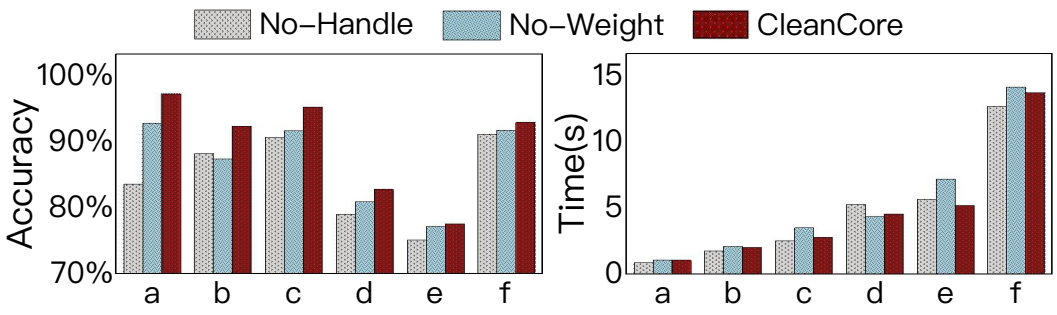


Figure 10: Ablation results for error handling and reliability weighting in CleanCore. Labels a–f denote Banknote, Seismic Bumps, Pendigits, Magic, SmartFactory, and Sensorless, respectively.

The complete CleanCore achieves the highest Accuracy on all six datasets, improving over No-Handle by 1.76–13.45 and over No-Weight by 0.38–4.84 percentage points. This shows that **error handling and reliability weighting help improve predictive performance and perform best together.**

The overall runtime costs of the three settings are similar, and complete CleanCore is even faster on some datasets. This shows that both mechanisms are lightweight and can promote faster convergence in some cases by reducing ineffective updates caused by erroneous samples. Overall, both bring stable performance improvements while maintaining low runtime costs.

Furthermore, we compare No-FER, Static-Core, and the complete CleanCore to examine the roles of feature repair and dynamic coreset maintenance. No-FER disables feature repair and downweights the corresponding candidate samples. Static-Core fixes the members of the initially constructed coreset and the sample mappings. Subsequent stages still perform error handling, reliability updating, and coreset weight aggregation, but no longer update the coreset members or mappings. Figure 11 shows the results.

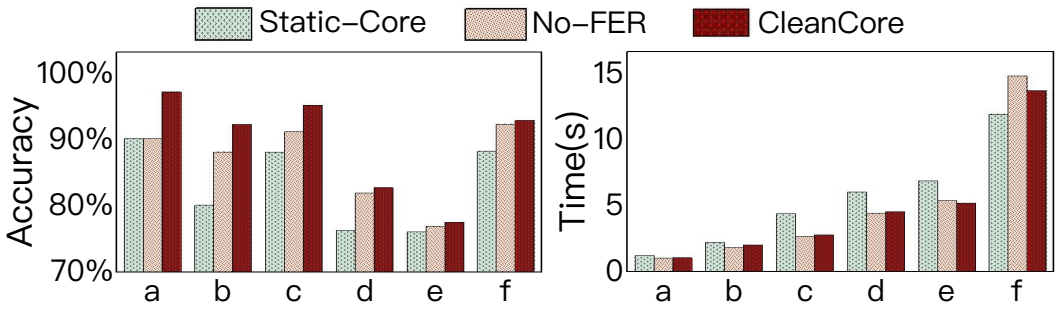


Figure 11: Comparison results for feature repair and dynamic coreset maintenance in CleanCore. Labels a–f denote Banknote, Seismic Bumps, Pendigits, Magic, SmartFactory, and Sensorless, respectively.

The complete CleanCore achieves the highest Accuracy on all six datasets, improving over No-FER by 0.55–6.91 and over Static-Core by 1.42–11.99 percentage points. These results provide empirical support for the roles of feature repair and dynamic coreset maintenance. Feature repair can directly adjust anomalous features, complementing the reduction of their influence through downweighting alone. Dynamic coreset maintenance allows representative samples and mappings to be adjusted as the training state and data quality change, helping maintain coreset representativeness.

In terms of runtime, the total costs of the complete CleanCore and the two variants are of a similar magnitude. Specifically, CleanCore has a lower total runtime than Static-Core on five datasets and than No-FER on SmartFactory and Sensorless. Total runtime depends not only on repair and maintenance costs but also on subsequent model training costs. Feature repair and dynamic coreset maintenance can reduce ineffective training by reducing error interference and maintaining coreset

representativeness, allowing the training time saved to offset or even exceed the additional processing costs. Therefore, the complete CleanCore achieves higher Accuracy while maintaining similar runtime, and is even faster on some datasets.

E.5 Supplementary Experiments

(1) **Effect Analysis of ErrorTyping.** To examine the typing reliability discussed in Theorem 1, we inject 20% random errors into the training sets of WDBC, Pendigits, Sensorless, MiniBooNE, and Skin, using pre-injection data as the reference. We measure the proportion of correct decisions among candidate samples actually subjected to typing in the first three executions of ErrorTyping. Figure 12(a) shows the results.

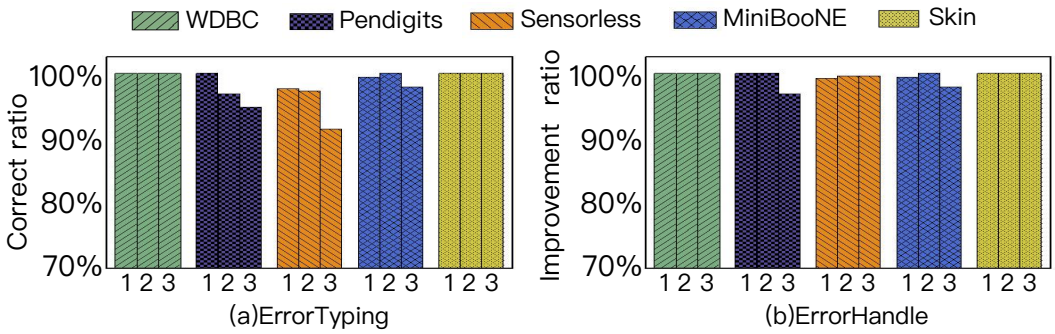


Figure 12: Effect analysis of ErrorTyping and ErrorHandle. Horizontal-axis values 1–3 denote the first three execution stages.

As Figure 12(a) shows, correct-decision proportions exceed 91% on all five datasets across the three stages. WDBC and Skin reach 100%, while Pendigits and MiniBooNE remain at or above 94.8% and 97.9%, respectively. Sensorless decreases as stages progress but remains at or above 91.4%. ErrorTyping thus maintains high and relatively stable correct-decision proportions, supporting subsequent targeted handling and providing supplementary empirical evidence for the reliability discussed in Theorem 1.

(2) **Effect Analysis of ErrorHandle.** To examine the error-handling bias control discussed in Theorem 2, we use the same datasets and error settings as Appendix E.5 (1) and measure the proportion of samples whose weighted gradient residual decreases after handling in ErrorHandle's first three executions. Each before-and-after comparison fixes the model parameters and uses the pre-injection sample gradient as the clean reference. The weighted gradient residual is the norm of the difference between the sample and clean-reference gradients multiplied by its reliability weight. This metric reflects the joint suppression of error influence by sample correction and reliability weighting. Figure 12(b) presents the results.

As Figure 12(b) shows, the proportion with decreased weighted residuals is at least 96.9% on every dataset across all three stages. WDBC and Skin reach 100%; Pendigits, Sensorless, and MiniBooNE remain at or above 96.9%, 99.2%, and 97.9%, respectively. ErrorHandle thus consistently reduces the weighted gradient residuals of most evaluated samples, reducing the effects of erroneous samples on model updates and providing supplementary empirical evidence for the bias-control analysis in Theorem 2.

(3) **Threshold Sensitivity Analysis.** To examine the parameter sensitivity of the threshold-setting strategy in Appendix D.4 (2), we select typical representatives from both decision and trigger thresholds, analyzing the high-loss screening quantile, loss-variability screening quantile, and coreset rebuilding threshold. The first two determine decision thresholds, while the last is a trigger threshold. We inject 20% random errors into the training sets of Seismic Bumps, Sensorless, MiniBooNE, and Skin; parameter values are shown in Figure 13.

As Figure 13(a, b) shows, varying the high-loss and loss-variability screening quantiles produces Accuracy ranges of 0.22–1.20 and 0.08–0.85 percentage points, respectively, across the four datasets,

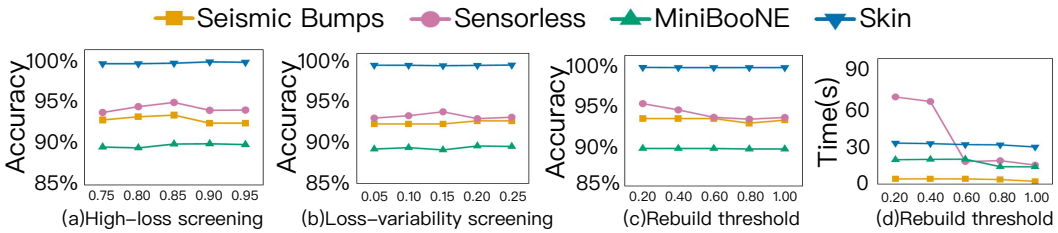


Figure 13: Threshold sensitivity of CleanCore. Horizontal axes denote the high-loss screening quantile (a), loss-variability screening quantile (b), and coreset rebuilding threshold (c, d).

indicating small overall changes. CleanCore therefore maintains good predictive performance across a broad range of reasonable thresholds, experimentally supporting the threshold-setting tolerance discussed in Appendix D.4 (2).

As Figure 13(c, d) shows, predictive performance remains relatively stable as the rebuilding threshold changes. More specifically, a smaller threshold triggers rebuilding more readily, helping maintain representativeness and obtain higher predictive performance but increasing runtime costs. A larger threshold reduces rebuilding costs but may cause some performance decline. This agrees with Appendix D.4 (2); in practice, performance and efficiency requirements can guide the trade-off.

(4) **Random Sampling Control Experiment.** To examine the effect of training subset selection on predictive performance, we compare Random and the complete CleanCore using MLP and tabular ResNet on the same eleven datasets as the main experiment. The number of samples in each control subset is the same as that in the coreset of the corresponding CleanCore run, and the CleanCore results are taken from the main experiment. Table 6 presents ACC, F1, and their improvements.

Table 6: Predictive performance comparison between Random and CleanCore. ACC and F1 are percentages; Δ denotes the improvement of CleanCore over Random in percentage points.

(a) MLP

ACC (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Random	93.04	92.73	79.69	87.82	79.29	75.30	86.35	82.18	98.20	71.60	76.24
CleanCore	99.13	100.00	93.42	96.66	85.52	77.75	94.53	90.04	99.73	82.48	80.09
Δ ACC	**+6.09**	**+7.27**	**+13.73**	**+8.83**	**+6.23**	**+2.45**	**+8.19**	**+7.86**	**+1.53**	**+10.87**	**+3.85**

F1 (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Random	92.57	92.62	47.84	87.80	76.50	50.51	86.28	77.79	97.35	47.06	76.09
CleanCore	99.07	100.00	59.65	96.71	83.43	57.54	94.52	87.65	99.59	69.70	79.65
Δ F1	**+6.50**	**+7.38**	**+11.81**	**+8.91**	**+6.93**	**+7.03**	**+8.24**	**+9.86**	**+2.24**	**+22.65**	**+3.57**

(b) Tabular ResNet

ACC (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Random	91.30	94.18	88.78	89.31	78.63	73.36	84.83	84.29	98.09	70.54	62.53
CleanCore	99.13	100.00	93.23	96.34	84.97	77.73	94.65	89.85	99.86	82.36	80.12
Δ ACC	**+7.83**	**+5.82**	**+4.45**	**+7.03**	**+6.33**	**+4.38**	**+9.82**	**+5.56**	**+1.77**	**+11.82**	**+17.59**

F1 (%) ↑

Method	WDBC	Banknote	Seismic Bumps	Pendigits	Magic	SmartFactory	Sensorless	MiniBooNE	Skin	Covertype	SUSY
Random	90.52	94.10	48.69	89.35	76.76	47.49	84.52	81.17	97.12	45.71	62.20
CleanCore	99.07	100.00	55.55	96.38	82.99	56.50	94.63	87.47	99.79	70.71	79.76
Δ F1	**+8.54**	**+5.90**	**+6.87**	**+7.03**	**+6.23**	**+9.01**	**+10.10**	**+6.30**	**+2.67**	**+25.01**	**+17.56**

In the twenty-two model–dataset settings currently listed, CleanCore achieves higher ACC and F1 in every setting, with improvements of 1.53–17.59 and 2.24–25.01 percentage points, respectively. On Covertype, the F1 improvements with MLP and tabular ResNet are 22.65 and 25.01 percentage points, respectively. In these comparisons, CleanCore achieves clear predictive-performance im-

provements over Random with the same subset size, providing supplementary empirical support for its design that combines error handling, reliability weighting, and coreset selection.