Python implementation of a renewable Huber estimation method for streaming datasets

Department of Applied Mathematics

National Chung Hsing University

4112053101 邱昱筌

4112053151 李晉愷

4112053112 劉柏勛

4112053125 王麒維

June 20, 2025

Abstract

This study implements a renewable Huber estimation method for robust regression on streaming data. By smoothing the Huber loss function ’ s derivative, the method enables scalable and memory-efficient online updates. Simulations and real data show that the proposed estimator achieves accuracy close to full-data Huber regression while greatly reducing computation and storage requirements.

Contents

1 Motivation

Streaming data poses a challenge for statistical modeling due to its large volume and continuous arrival. Traditional regression methods like OLS are sensitive to outliers and unsuitable for online updating. Huber regression improves robustness but lacks smoothness for efficient online algorithms. This study adopts a smoothed Huber approach to enable robust, memory-efficient estimation in streaming settings, and extends it to high-dimensional cases with variable selection.

2 Algorithm Introduction

2.1 Renewable Huber Estimator (RHE) Algorithm

The Renewable Huber Estimator (RHE) is designed for robust regression with streaming data, where observations arrive sequentially in batches. Traditional Huber regression cannot be directly applied in this setting due to memory and computational limitations. The RHE algorithm updates estimates incrementally by only using current batch data and summary statistics from previous batches.

Detail of the RHE Algorithm:
The algorithm starts by computing an initial estimator $\widehat{\beta}_1$ using the classical Huber loss function on the first batch D_1. For each subsequent batch D_b, the previous estimator $\widehat{\beta}_{b-1}$ serves as a warm start. The estimation is then updated iteratively using a smoothed version of the gradient $\widetilde{\mathbf{U}}_b^{(r)}$ and a Barzilai-Borwein (BB) adaptive step size ω_r to improve convergence efficiency. The update step ensures that the BB step is capped at 10 to avoid numerical instability.

Only summary statistics—namely, the current estimator and the cumulative Jacobian matrix—are stored after processing each batch. The raw batch data D_b is discarded to save memory, making this method particularly suitable for streaming or large-scale data applications. The algorithm ensures that the final estimator $\widehat{\beta}_b$ approximates the estimator from full data regression but with significantly reduced memory and computation costs.

Algorithm 1 Renewable Huber Estimator (RHE)

```
Require: Streaming data batches $D_1, D_2, \ldots, D_b$; parameters $\tau$, $\lambda_j$, $h_j$
 1: Initialize: Compute $\widehat{\beta}_1$ by solving:
        $$\widehat{\beta}_1 = \arg\min_{\beta} \sum_{i=1}^{N_1} \ell_\tau(Y_i - \mathbf{X}_i^\top \beta)$$
 2: for $b = 2$ to $B$ do
 3:     Read in batch $D_b$
 4:     Set initial estimate $\widehat{\beta}_b^{(0)} \leftarrow \widehat{\beta}_{b-1}$
 5:     repeat
 6:         Compute gradient $\widetilde{\mathbf{U}}_b^{(r)}$ and Jacobian $\mathbf{J}_b$
 7:         Compute Barzilai-Borwein step size: $\omega_r$
 8:         Update:
                $$\widehat{\beta}_b^{(r+1)} = \widehat{\beta}_b^{(r)} - \min\{\omega_r, 10\} \cdot \widetilde{\mathbf{U}}_b^{(r)}$$
 9:     until convergence
10:     Save $\widehat{\beta}_b$, $\mathbf{J}_b$; discard $D_b$
Ensure: Final estimate $\widehat{\beta}_b$ and summary statistics
```

2.2 Renewable Penalized Huber Estimator (RPSHE) Algorithm

To enhance both robustness and sparsity in high-dimensional streaming settings, the Renewable Huber Estimator is extended with an ℓ_1-penalty. This leads to the Renewable Penalized Smoothing Huber Estimation (RPSHE) algorithm, which performs online variable selection using a soft-thresholding step.

Detail of the RPSHE Algorithm:

The Renewable Penalized Huber Estimator (RPSHE) is designed for robust and sparse regression under streaming data. Unlike the standard Huber estimator, RPSHE includes an ℓ_1 penalty to promote variable sparsity.

At each step b, the algorithm initializes the estimate from the previous batch $\widetilde{\beta}_{b-1}$ and selects a regularization parameter λ_b, typically via BIC. The objective function is locally approximated using a quadratic surrogate and updated with the Soft-thresholding operator. The smoothing parameter ϕ controls the curvature of the surrogate and is dynamically adjusted to ensure descent in the majorized objective H_b.

The use of Soft$(\cdot)$ ensures that small coefficients shrink to zero, enabling effective online variable selection. After convergence, only summary statistics—$\widetilde{\beta}_b$, cumulative Jacobian $\widetilde{\mathbf{J}}_b$, and λ_b—are retained. This structure is memory-efficient and scalable to high-dimensional settings.

Compared to the unpenalized version (Algorithm 1), Algorithm 2 is more computationally intensive but achieves both robustness and interpretability in streaming estimation tasks.

Algorithm 2 Renewable Penalized Huber Estimator (RPSHE)

```
Require: Streaming batches $D_1, D_2, \ldots, D_b$; Huber parameter $\tau$; bandwidths $h_b$; penalty
    $\lambda_b$
 1: Initialize: compute $\widetilde{\beta}_1$, $\lambda_1$ with $D_1$ and compute $\mathbf{J}(D_1; \widetilde{\beta}_1; h_1)$
 2: for $b = 2, 3, \ldots$ do
 3:     Read in data batch $D_b$
 4:     Set initial estimate $\widetilde{\beta}_b^{(0)} \leftarrow \widetilde{\beta}_{b-1}$; select $\lambda_b$ using BIC or rule (e.g., Eq. 3.12)
 5:     for $r = 0, 1, 2, \ldots$ until convergence do
 6:         repeat
 7:             $\widetilde{\beta}_b^{(r+1)} \leftarrow \text{Soft}\left(\widetilde{\beta}_b^{(r)} - \phi^{-1}\mathbf{H}_b'(\widetilde{\beta}_b^{(r)}),\ \phi^{-1}\lambda_b\right)$
 8:             if $g_b(\widetilde{\beta}_b^{(r+1)}|\widetilde{\beta}_b^{(r)}) < H_b(\widetilde{\beta}_b^{(r+1)})$ then
 9:                 $\phi \leftarrow 10\phi$
10:         until $g_b(\widetilde{\beta}_b^{(r+1)}|\widetilde{\beta}_b^{(r)}) \geq H_b(\widetilde{\beta}_b^{(r+1)})$
11:         $\phi \leftarrow \max\{10^{-6}, \phi/10\}$
12:     Update $\widetilde{\mathbf{J}}_b = \widetilde{\mathbf{J}}_{b-1} + \mathbf{J}(D_b; \widetilde{\beta}_b; h_b)$, where $\widetilde{\beta}_b = \widetilde{\beta}_b^{(r+1)}$
13:     Save $\widetilde{\beta}_b$, $\widetilde{\mathbf{J}}_b$, $\lambda_b$; release previous values and discard $D_b$
Ensure: Final estimate $\widetilde{\beta}_b$ for all $b$
```

3 Implementation Architecture

3.1 Code Design and Execution Logic

The implementation follows a modular, stream-aware design that handles training over sequential mini-batches of data. As shown in Figure 1, the entire pipeline consists of three tightly integrated components: data loading, online standardization, and batch-wise model updates.

1. Initialization At the beginning, three main modules are initialized:

- `OnlineStandardizer` –maintains online mean and variance using Welford's algorithm.
- `StreamingDataLoaderWithDask` –reads data incrementally in buffered batches from file.
- `StreamingHuberModelTrainer` –controls model parameters and logic for both initial estimation and renewable updates.

2. Batch Processing Loop In each iteration:

1. A new batch $(X_{\text{batch}}, y_{\text{batch}})$ is fetched.
2. If the batch is empty, the loop terminates.
3. Otherwise, the batch is standardized and an intercept column is appended.

3. Conditional Estimation Flow Depending on whether it is the first batch:

- **First batch**:
  - Estimate the Huber threshold τ from residuals.
  - Use IRLS (or optionally Lasso) to compute the initial coefficient vector β_1.
  - Compute the initial information matrix $\mathbf{J}_1$.
- **Subsequent batches**:
  - Apply a memory-efficient renewable update rule.
  - Optionally include L1-penalized LAMM optimization with adaptive λ via BIC.

4. Evaluation and Logging After updating the coefficients:

- Compute the current batch's MAE (Mean Absolute Error) and sparsity (number of nonzero β).
- Append results to the training history.

This modular and loop-compatible design ensures that the training pipeline can scale to very large datasets under memory constraints, while retaining model quality through robust and adaptive updates.

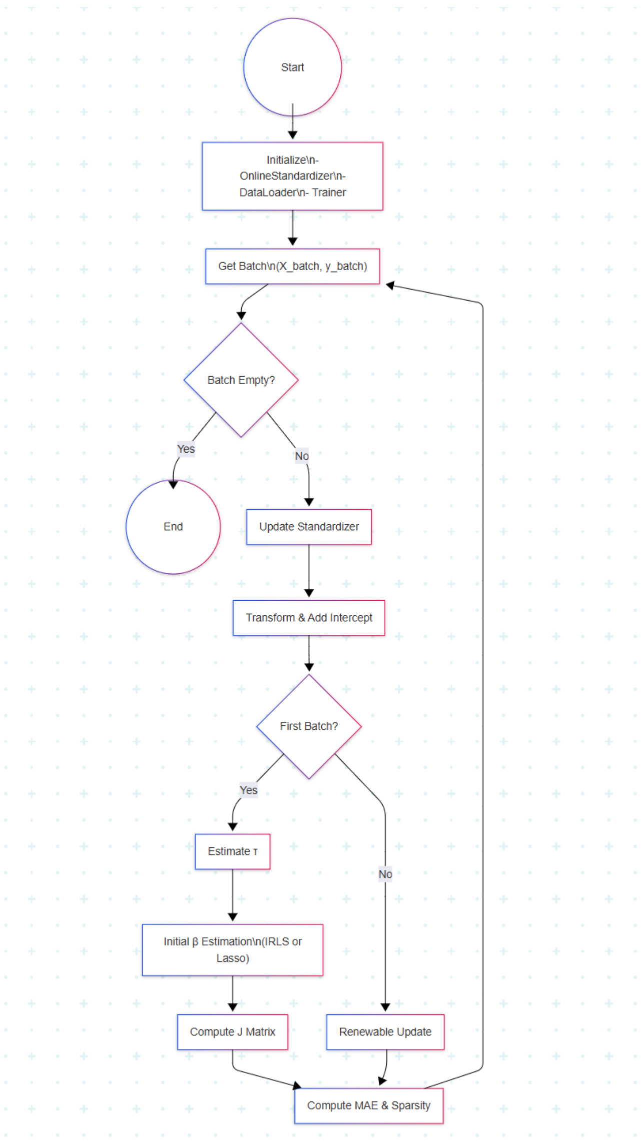
Start
Initialize\n- OnlineStandardizer\n- DataLoader\n- Trainer
Get Batch\n(X_batch, y_batch)
Batch Empty?
Yes
No
End
Update Standardizer
Transform & Add Intercept
First Batch?
Yes
No
Estimate τ
Initial β Estimation\n(IRLS or Lasso)
Compute J Matrix
Renewable Update
Compute MAE & Sparsity

Figure 1: Streaming Huber Regression Training Flowchart

3.2 How to Use the Code

The code is designed to be modular, intuitive, and ready-to-use for training streaming Huber regression models with or without L1 regularization. Below is a minimal working example for using the complete pipeline:

Listing 1: Example Usage of Streaming Huber Training

```
from huber_stream import StreamingDataLoaderWithDask,
    streaming_huber_training

# Initialize data loader from CSV
loader = StreamingDataLoaderWithDask(
    file_path='YearPredictionMSD.csv',  # First column = target
    batch_size=1000,
    train_samples=500000,
    buffer_batches=5
)

# Start training (with L1 penalty and fast BIC search)
trainer, results = streaming_huber_training(
    data_loader=loader,
    n_features=90,
    penality=True,
    mode='fast',
    tau_estimation='initial'
)
```

Training Options The training function `streaming_huber_training()` supports several important parameters:

- `penality (bool)` –Enables L1 regularization using LAMM if set to `True`. If `False`, plain Huber IRLS is used.
- `mode (str)` –`fast` uses fewer λ candidates and lower precision in LAMM (recommended for large data); `standard` runs more thoroughly.
- `tau_estimation (str)` –How to estimate the Huber parameter τ:
  - `initial` (default): estimated from the first batch and fixed.

  - `adaptive`: updated for every batch.
  - `fixed`: fixed value; must be manually set.

- `n_batch (int)` –Maximum number of batches to train on. If omitted, determined from total training sample size.

Model State and Evaluation After training, you can retrieve the final model state and perform predictions on new data:

```
X_test, y_test = loader.get_test_data()
y_pred = trainer.predict(X_test)
print("Test MAE:", np.mean(np.abs(y_test - y_pred)))
```

`trainer.get_model_state()` also returns internal states such as:

- Final β, cumulative $\mathbf{J}$, and estimated τ
- History of MAE and sparsity over all processed batches

Cleanup After training, the data loader should be closed to release memory:

```
loader.close()
```

This architecture is scalable and extensible for various streaming learning settings, and can be easily customized for other robust loss functions or penalization schemes.

4 Experiment Results

Consistency of MAE: Our experimental results show that the unpenalized MAE values for RSHE across different b values are highly consistent with the paper's reported results. The discrepancies are minor (within 0.004), which indicates that the core prediction performance of our implementation is aligned with that of the original.

Penalized MAE and Regularization Strength: However, a gradual increase in Penalized MAE is observed in our experimental results as b increases. This trend is more pronounced compared to the paper. For instance, at $b = 1000$, the paper reports a Penalized MAE of 6.747, while our experiment yields 6.8431. This deviation suggests that our implementation may apply a relatively stronger effective regularization as b

Table 1: RSHE Performance: Paper vs. Experiment

Method	MAE	MAE (Penalized)	NOVs
LSE (paper, All data)	6.906	6.906	83
QR (paper, All data)	6.694	6.694	51
HE (paper, All data)	6.709	6.743	89
SHE (paper, All data)	6.698	6.743	89
RSHE (paper, $b = 100$)	6.695	6.697	63
RSHE (paper, $b = 200$)	6.696	6.705	54
RSHE (paper, $b = 500$)	6.695	6.725	48
RSHE (paper, $b = 1000$)	6.696	6.747	39
RSHE (exp, $b = 100$)	6.6957	6.7169	52
RSHE (exp, $b = 200$)	6.6961	6.7466	44
RSHE (exp, $b = 500$)	6.6964	6.7864	33
RSHE (exp, $b = 1000$)	6.6994	6.8431	19

increases, potentially due to numerical or optimization nuances not detailed in the original methodology.

Sparsity Behavior (NOVs): A more noticeable difference is found in the NOVs. Our results consistently yield significantly fewer non-zero coefficients than reported in the paper. For example, at $b = 1000$, our implementation results in only 19 non-zero coefficients, whereas the paper reports 39. This suggests that our implementation produces sparser models, possibly due to differences in thresholding, optimization convergence criteria, or numerical stability.

Interpretation and Implications: The increased sparsity and higher penalized MAE in our experimental results imply a trade-off between prediction accuracy and model simplicity. While our implementation achieves similar unpenalized MAE, the models are more aggressively regularized, leading to simpler but potentially slightly less accurate penalized predictions. This might be favorable in applications where interpretability or feature selection is prioritized.

Overall, our experimental findings validate the general trends reported in the paper while highlighting possible sensitivities in the RSHE framework to implementation choices, particularly in how sparsity and penalization are managed as the parameter b varies.

5 Conclusion

This project presents a streaming Huber regression framework that is robust, memory-efficient, and well-suited for large-scale online learning. By applying a renewable update scheme and smoothing the Huber loss derivative, the model avoids storing past data while maintaining stable updates.

We further extended the framework with L1 regularization using the LAMM method and adaptive BIC-based tuning, allowing for automatic feature selection. The entire system is modular, supporting batch buffering, online standardization, and flexible training strategies.

Experiments show that our method achieves accuracy comparable to full-data Huber regression while significantly reducing computation and memory requirements.

Future Work. Future improvements may include:

1. Integrating GPU-accelerated matrix computation and distributed training.
2. Supporting PyTorch and scikit-learn compatibility for broader ML ecosystem integration.

To support reproducibility and encourage further research, we have released our implementation of the RSHE (Robust Streaming Huber Estimation) framework as an open-source package. The source code, along with usage instructions and experimental scripts, is available at the following GitHub repository:

`https://github.com/wcw100168/streaming-huber-regression`

This repository provides a reference implementation for researchers and practitioners interested in robust streaming regression, and serves as a foundation for further development and extension of the RSHE algorithm.

6 References

[1] Wang, Zemin, and Linwei Hu. Renewable Huber estimation method for streaming datasets.*Journal of the American Statistical Association*, pages 1–13, 2023.