CLC _____ Number _____

UDC _____ Available for reference ☐Yes ☐No

SUSTech Southern University of Science and Technology

Undergraduate Thesis

Thesis Title: **Code Reproduction of Online Debiased Lasso for Streaming Data**

Student Name: **Hanbin Liu**

Student ID: **11912410**

Department: **Department of Statistic and Data Science**

Program: **Statistics**

Thesis Advisor: **Bingyi Jing**

Date: May 8, 2023

COMMITMENT OF HONESTY

1. I solemnly promise that the paper presented comes from my independent research work under my supervisor's supervision. All statistics and images are real and reliable.

2. Except for the annotated reference, the paper contents no other published work or achievement by person or group. All people making important contributions to the study of the paper have been indicated clearly in the paper.

3. I promise that I did not plagiarize other people's research achievement or forge related data in the process of designing topic and research content.

4. If there is violation of any intellectual property right, I will take legal responsibility myself.

Signature:

Date:

Code Reproduction of Online Debiased Lasso for Streaming Data

Hanbin Liu

Department of Statistic and Data Science Tutor: Bingyi Jing

[**ABSTRACT**]: I accomplish the code reproduction of the online debiased lasso (ODL) algorithm. About this thesis, first, a brief introduction to streaming data and online learning is made at the beginning. Then I summarize a paper that proposes the ODL algorithm, whose properties motivate me to conduct a code reproduction. To conduct this, the principle and framework of the ODL algorithm are described in detail according to that paper, added some of my error corrections and comments on that paper. Mainly, I complete the code reproduction of the ODL algorithm basically consistently and find some unclear points and data problems in the algorithm implementation in the original paper based on the simulation results. Further, I contact the author to confirm and address these issues.

[**Key words**]: Sreaming data; Online learning; Debiased lasso; Online lasso

Table of Content

1. Indtroduction

1.1 Streaming data

The exponential growth of data generated by businesses and individuals has led to the emergence of streaming data. Streaming data is a continuous flow of data generated and processed in real-time, without storing the data in a traditional database or file system. Streaming data is used in various industries, including finance, healthcare, manufacturing, and transportation, for real-time monitoring, predictive maintenance, and risk management.

One way to represent streaming data mathematically is through a data stream model. The data stream model assumes that the data arrives in a sequential and continuous manner, and that it is not possible to store all the data in memory. Therefore, the model uses a limited amount of memory to process the data, and discards old data as new data arrives.

Mathematically, the data stream model can be represented as follows:

Let S be a data stream of n data points. The data points are represented as $\mathcal{D} = \{d_1, d_2, d_3, ..., d_n\}$, where each data point d_i is composed of m attributes or features, denoted as $\boldsymbol{x}_i = \{x_{1i}, x_{2i}, x_{3i}, ..., x_{mi}\}$. Let W be a window of size k that slides over the data stream, and let $R(W)$ be the result of processing the data within the window. The data stream model is defined as follows:

$R(W) = f(\mathcal{D}_{i-k+1}, \mathcal{D}_{i-k+2}, ..., \mathcal{D}_i)$, where $\mathcal{D}_i$ is the i-th data point in the data stream, and f is a function that processes the data within the window.

This mathematical model represents the basic idea behind processing streaming data: the data is processed within a window of limited size, and the result of the processing is based on the data within that window. The window slides over the data stream, and as new data points arrive, old data points are discarded.

1.2 Online learning

Online learning, also known as incremental learning or lifelong learning, is a machine learning technique in which the model is trained on data that arrives sequentially or in a stream, rather than in a batch. The model learns from each new observation and updates its

parameters incrementally. This is in contrast to batch learning, where the model is trained on a fixed dataset and cannot be updated until the next batch of data arrives.

Mathematically, the goal of online learning is to minimize a loss function over a sequence of observations. Let $\boldsymbol{x}_i$ be the input feature vector for the i-th observation, and $\boldsymbol{y}_i$ be its corresponding target variable. Let $f(\boldsymbol{x};\boldsymbol{\theta})$ be the model, where $\boldsymbol{\theta}$ is a set of learnable parameters. The loss function $\mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{\theta}),\boldsymbol{y}_i)$ measures the discrepancy between the predicted output of the model and the true target value. The goal of online learning is to find the parameter vector $\boldsymbol{\theta}$ that minimizes the expected loss:

$$\min_{\boldsymbol{\theta}} E_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{F}}\Big[\mathcal{L}(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y})\Big]$$

where $\mathcal{F}$ is the data distribution.

Online learning is particularly useful in scenarios where the data is large, high-dimensional, or arriving continuously, such as in natural language processing, computer vision, and sensor networks. Online learning allows the model to adapt to new and changing data without having to retrain the entire model on a new dataset. It also enables real-time decision-making and can reduce computational costs by updating the model incrementally.

Overall, online learning is an important technique in machine learning that has numerous applications in a variety of fields. Its mathematical formulation provides a rigorous framework for optimizing the model parameters and making accurate predictions on streaming data.

1.3 Motivation and target paper

The whole work is mainly based on Han et al.[1] (version on arXiv: v2). That is, the algorithm and data for code reproduction come from this paper.

Han et al.[1] describes a new statistical method called Online Debiased Lasso (ODL) that is designed to analyze high-dimensional streaming data. The proposed approach utilizes online lasso estimation and online debiased lasso instead of accessing the entire dataset, only requiring the availability of the current data batch and sufficient statistics of historical data at

each stage of the analysis. Additionally, ODL proposes an adaptive procedure to determine and update the tuning parameter λ dynamically upon the arrival of a new data batch. This feature enables the adjustment of the regularization amount along with data accumulation.

Han et al.[1] provides a detailed theoretical analysis of the proposed ODL estimator, including its asymptotic normality under certain conditions. These results provide a theoretical basis for constructing confidence intervals and conducting hypothesis tests. Extensive numerical experiments with simulated data demonstrate the computational efficiency of the ODL algorithm and its strong support for the theoretical properties of the ODL estimator.

Han et al.[1] has several main sections. Section 1 introduces the purpose of the paper, while Section 2 presents the model formulation and the proposed ODL procedure. Section 3 includes the theoretical properties of the proposed ODL estimator. Simulation experiments are given in Section 4 to evaluate the performance of ODL in comparison to the offline ordinary least square estimator.

My work focused on the code reproduction of the ODL (no code is available on arXiv or Github until March 2023) algorithm and some simulation tests according to Han et al.[1]. Besides, I find some errors in Han et al.[1] and communicate with the author (Ruijian Han) to solve these issues. The following section describes the framework and process of ODL based on Han et al.[1] and other references.

2. Online debised lasso

This entire section describes the principles and framework of ODL mainly based on Han et al.[1], but some of my findings and interpretations are added where necessary.

2.1 Offline debiased lasso

Consider b data batches arriving in a sequence, denoted by $\{\mathcal{D}_1, ..., \mathcal{D}_b\}$, where $b \geqslant 2$ is the time point and the number of total samples is N_b. A high-dimensional linear model is considered here, that is, for each data batch $\mathcal{D}_j = \{\boldsymbol{y}^{(j)}, \boldsymbol{X}^{(j)}\}$, let

$$\boldsymbol{y}^{(j)} = \boldsymbol{X}^{(j)}\boldsymbol{\beta}_0 + \boldsymbol{\epsilon}^{(j)},\ j = 1, ..., b, \tag{1}$$

where $\boldsymbol{y}^{(j)} = (y_1^{(j)}, ..., y_{n_j}^{(j)})^\top$ is the vector of dependent variables, $\boldsymbol{X}^{(j)} = (x_1^{(j)}, ..., x_{n_j}^{(j)})^\top$ is the design matrix with size $n_j \times p$, n_j is the data batch size for $\mathcal{D}_j$ and $p \geqslant N_b = \sum_{j=1}^{b} n_j$ is the dimension. Besides, $\boldsymbol{\beta}_0 = (\beta_{0,1}, ..., \beta_{0,p})^\top \in \mathbb{R}^p$ is the regression coefficient which is unknown but sparse because of the high-dimensional assumption, and $\epsilon_i^{(j)},\ i = 1, ..., n_j$ are i.i.d error terms with mean 0 and finite variance σ_ϵ^2.

To reduce weight in a streaming data scenario where the data volume increases rapidly over time, it may not be feasible to store individual-level raw data for an extended period. As a result, it is difficult to apply offline debiased algorithms that require access to the whole dataset such as those proposed by Zhang et al.[2], van de Geer et al.[3], and Javanmard et al.[4]. To overcome this problem, Han et al.[1] introduces an online debiasing process to estimate $\beta_{0,r}$, the r-th component of $\boldsymbol{\beta}_0$, where $r = 1, ..., p$ in (11).

Initially, when the first data batch $\mathcal{D}_1 = \{\boldsymbol{y}^{(1)}, \boldsymbol{X}^{(1)}\}$ arrives, Han et al.[1] uses the offline debiased lasso method to obtain an estimator by considering the r-th column of $\boldsymbol{X}^{(1)}$ as $\boldsymbol{x}_r^{(1)}$ and the sub-matrix of $\boldsymbol{X}^{(1)}$ that excludes the r-th column as $\boldsymbol{X}_{-r}^{(1)}$. Let $\lambda_1 \geqslant 0$ be the regularization parameter, then the initial offline lasso estimator for $b = 1$ is given by

$$\widehat{\boldsymbol{\beta}}^{(1)} := \underset{\boldsymbol{\beta} \in \mathbb{R}^p}{\arg\min} \left\{ \frac{1}{2n_1} ||\boldsymbol{y}^{(1)} - \boldsymbol{X}^{(1)}\boldsymbol{\beta}||_2^2 + \lambda_1 ||\boldsymbol{\beta}||_1 \right\}, \tag{2}$$

To construct a confidence interval for $\beta_{0,r}$, by Zhang et al.[2], a low-dimensional vector $\widehat{\boldsymbol{z}}_r^{(1)}$ is needed. Here, $\widehat{\boldsymbol{z}}_r^{(1)}$ can act as the projection of $\boldsymbol{x}_r^{(1)}$ to the orthogonal complement of the column space of $\boldsymbol{X}_{-r}^{(1)}$. Specifically, $\widehat{\boldsymbol{z}}_r^{(1)} := \boldsymbol{x}_r^{(1)} - \boldsymbol{X}_{-r}^{(1)}\widehat{\boldsymbol{\gamma}}_r^{(1)}$, where

$$\widehat{\boldsymbol{\gamma}}_r^{(1)} := \underset{\boldsymbol{\gamma} \in \mathbb{R}^{(p-1)}}{\arg\min} \left\{ \frac{1}{2n_1} ||\boldsymbol{x}_r^{(1)} - \boldsymbol{X}_{-r}^{(1)}\boldsymbol{\gamma}||_2^2 + \lambda_1 ||\boldsymbol{\gamma}||_1 \right\}. \tag{3}$$

and λ_1 is selected to be the same as in (2). Han et al.[1] then obtains the offline debiased lasso estimator of $\boldsymbol{\beta}_0,\ r = 1, ..., p$ by

$$\widehat{\beta}_{\text{off},r}^{(1)} := \widehat{\beta}_r^{(1)} - \frac{(\widehat{\boldsymbol{z}}_r^{(1)})^\top (\boldsymbol{y}^{(1)} - \boldsymbol{X}^{(1)}\widehat{\boldsymbol{\beta}}^{(1)})}{(\widehat{\boldsymbol{z}}_r^{(1)})^\top \boldsymbol{x}_r^{(1)}}.$$

However, this is wrong. I contacted the author (Ruijian Han) and confirmed with him that

the correct expression should be

$$\widehat{\beta}^{(1)}_{\text{off},r} := \widehat{\beta}^{(1)}_r + \frac{(\widehat{\boldsymbol{z}}^{(1)}_r)^\top (\boldsymbol{y}^{(1)} - \boldsymbol{X}^{(1)}\widehat{\boldsymbol{\beta}}^{(1)})}{(\widehat{\boldsymbol{z}}^{(1)}_r)^\top \boldsymbol{x}^{(1)}_r}. \tag{4}$$

Back to the framework, in the offline debiased lasso algorithm, when the second data batch $\mathcal{D}_2 = \{\boldsymbol{y}^{(2)}, \boldsymbol{X}^{(2)}\}$ is received, the initial dataset $\{\boldsymbol{y}^{(1)}, \boldsymbol{X}^{(1)}\}$ is augmented with $\{\boldsymbol{y}^{(2)}, \boldsymbol{X}^{(2)}\}$ to replace the former. However, in an online setting, the initial dataset $\{\boldsymbol{y}^{(1)}, \boldsymbol{X}^{(1)}\}$ may not be available in the time point $b = 2$, which poses a challenge. Han et al.[1] provides an approach to tackle this issue by developing an online lasso estimation and low-dimensional projection process that only relies on the current data batch and summary statistics derived from historical raw data. The online section for ODL consists of several key subsections, which are outlined below.

2.2 Online lasso

For $b \geqslant 2$, if all previous data batches $\mathcal{D}_1, ..., \mathcal{D}_{b-1}$ are available when a new data batch $\mathcal{D}_b = \{\boldsymbol{y}^{(b)}, \boldsymbol{X}^{(b)}\}$ arrives, we can employ the offline lasso method to solve an optimization problem that involves minimizing the ℓ_1-penalized residual sum of squares. This is the ideal situation. Let $\mathcal{H}(\boldsymbol{\beta})$ be the goal function, then the estimator of $\boldsymbol{\beta}^{(b)}$ is given by

$$\widehat{\boldsymbol{\beta}}^{(b)}(\lambda_b) := \underset{\beta \in \mathbb{R}^p}{\arg\min}\, \mathcal{H}(\boldsymbol{\beta}) = \underset{\beta \in \mathbb{R}^p}{\arg\min} \left\{ \frac{1}{2N_b} \sum_{j=1}^{b} ||\boldsymbol{y}^{(j)} - \boldsymbol{X}^{(j)}\boldsymbol{\beta}||_2^2 + \lambda_b ||\boldsymbol{\beta}||_1 \right\}, \tag{5}$$

where the regularization parameter λ_b is adaptively selected for step b and $N_b = \sum_{j=1}^{b} n_j$ is the cumulative sample size. However, since only the availability of summary statistics of historical data is assumed, algorithms like coordinate descent (Friedman et al.[5]) that require the entire dataset cannot be used. While coordinate descent cannot work, gradient descent might work, though probably not as well in the lasso problem. If we assume that we do not have access to the entire dataset and use the gradient descent algorithm's iterative update formula, we can observe that the function $\mathcal{H}(\boldsymbol{\beta})$ depends only on some summary statistics

derived from the data. Specifically, the gradient of $\mathcal{H}(\boldsymbol{\beta})$ w.r.t $\boldsymbol{\beta}$ is given by

$$\frac{\partial \mathcal{H}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \frac{1}{N_b}\left((\sum_{j=1}^{b}(\boldsymbol{X}^{(j)})^\top \boldsymbol{X}^{(j)})\boldsymbol{\beta} - \sum_{j=1}^{b}(\boldsymbol{X}^{(j)})^\top \boldsymbol{y}^{(j)}\right) + \lambda_b \times \text{sgn}(\boldsymbol{\beta}).$$

Naturally, the summary statistics are defined as

$$\boldsymbol{S}^{(b)} = \sum_{j=1}^{b}(\boldsymbol{X}^{(j)})^\top \boldsymbol{X}^{(j)},\ \boldsymbol{U}^{(b)} = \sum_{j=1}^{b}(\boldsymbol{X}^{(j)})^\top \boldsymbol{y}^{(j)}. \tag{6}$$

By utilizing these summary statistics, one can find the solution to equation (5) through the use of a gradient descent algorithm. Specifically, let $\mathcal{L}(\boldsymbol{\beta})$ be the first term of $\mathcal{H}(\boldsymbol{\beta})$, then the gradient of $\mathcal{L}(\boldsymbol{\beta})$ w.r.t $\boldsymbol{\beta}$ is given by

$$\frac{\partial \mathcal{L}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \frac{1}{N_b}(\boldsymbol{S}^{(b)}\boldsymbol{\beta} - \boldsymbol{U}^{(b)}). \tag{7}$$

It is worth mentioning that the gradient computation is solely based on the summary statistics $\boldsymbol{S}^{(b)}$ and $\boldsymbol{U}^{(b)}$ derived from the historical raw data. This is the key step in the whole framework in ODL and this idea is also used in the low-dimensional projection when performing a lasso estimation. To obtain the solution to (5), Han et al.[1] uses a combination of gradient descent and soft thresholding (Daubechies et al.[6]; Donoho et al.[7]). The iterative process involves two steps:

(i) Gradient descent: let η be the learning rate, and update $\widehat{\boldsymbol{\beta}}^{(b)}$ by

$$\widehat{\boldsymbol{\beta}}^{(b)} \leftarrow \widehat{\boldsymbol{\beta}}^{(b)} - \eta\frac{\partial \mathcal{L}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \widehat{\boldsymbol{\beta}}^{(b)} - \frac{\eta}{N_b}(\boldsymbol{S}^{(b)}\widehat{\boldsymbol{\beta}}^{(b)} - \boldsymbol{U}^{(b)}). \tag{8}$$

(ii) Soft thresholding: let $\mathcal{S}(x, \lambda) = \text{sgn}(x)(|x| - \lambda)_+$, and apply $\mathcal{S}(\widehat{\beta}_r^{(b)}; \eta\lambda_b)$ to the r-th component in $\widehat{\boldsymbol{\beta}}^{(b)}$ in (i) for $r = 1, ..., p$.

The iterative process of gradient descent and soft thresholding continues until the stopping criterion, set as $||\partial \mathcal{L}(\boldsymbol{\beta})/\partial \boldsymbol{\beta}||_2 \leqslant 10^{-6}$, is met.

As more data batches are received, the size of the sufficient statistics remains constant, and the summary statistics are updated incrementally without increasing in size. This is

achieved by updating the matrices $\boldsymbol{S}^{(b)}$ and $\boldsymbol{U}^{(b)}$ upon the arrival of a new batch $\mathcal{D}_b$ by

$$\boldsymbol{S}^{(b)} = \boldsymbol{S}^{(b-1)} + (\boldsymbol{X}^{(b)})^\top \boldsymbol{X}^{(b)},\ \boldsymbol{U}^{(b)} = \boldsymbol{U}^{(b-1)} + (\boldsymbol{X}^{(b)})^\top \boldsymbol{y}^{(b)}.$$

Furthermore, an estimator of σ_ϵ^2 is obtained consistently using the method of moments and is utilized to construct the confidence intervals. Han et al.[1] obtains the estimator of σ_ϵ^2 by

$$(\widehat{\sigma}_\epsilon^2)^{(b)} = \frac{N_{b-1}}{N_b}(\widehat{\sigma}_\epsilon^2)^{(b-1)} + \frac{n_b}{N_b}(\boldsymbol{y}^{(b)} - \boldsymbol{X}^{(b)}\widehat{\boldsymbol{\beta}}^{(b)})^\top(\boldsymbol{y}^{(b)} - \boldsymbol{X}^{(b)}\widehat{\boldsymbol{\beta}}^{(b)}).$$

However, this is wrong. I contacted the author (Ruijian Han) and confirmed with him that the correct expression should be

$$(\widehat{\sigma}_\epsilon^2)^{(b)} = \frac{N_{b-1}}{N_b}(\widehat{\sigma}_\epsilon^2)^{(b-1)} + \frac{1}{N_b}(\boldsymbol{y}^{(b)} - \boldsymbol{X}^{(b)}\widehat{\boldsymbol{\beta}}^{(b)})^\top(\boldsymbol{y}^{(b)} - \boldsymbol{X}^{(b)}\widehat{\boldsymbol{\beta}}^{(b)}). \tag{9}$$

Besides, considering that it is unrealistic to use moment estimation in high-dimensional estimation to estimate variance, this expression does not give the estimator of σ_ϵ^2 for $b = 1$, and this will be a confusing issue when implementing the ODL algorithm. Further details are discussed in Section 3.

2.3 Online low-dimensional projection

Han et al.[1], then, develops an online estimator for the low-dimensional projection. Let the sample size be denoted byN_b, and λ_b be the regularization parameter as in (5). Then the online estimator for the low-dimensional projection is given by

$$\widehat{\boldsymbol{\gamma}}_r^{(b)} := \underset{\gamma \in \mathbb{R}^{(p-1)}}{\arg\min} \left\{ \frac{1}{2N_b} \sum_{j=1}^{b} ||\boldsymbol{x}_r^{(j)} - \boldsymbol{X}_{-r}^{(j)}\boldsymbol{\gamma}||_2^2 + \lambda_b ||\boldsymbol{\gamma}||_1 \right\}, \tag{10}$$

Since this is also a lasso estimation problem, similarly, the data information can be summarized by the statistics $\boldsymbol{R}^{(b)} = \sum_{j=1}^{b} (\boldsymbol{X}_{-r}^{(j)})^\top \boldsymbol{X}_{-r}^{(j)}$ and $\boldsymbol{T}^{(b)} = \sum_{j=1}^{b} (\boldsymbol{X}_{-r}^{(j)})^\top \boldsymbol{x}_r^{(j)}$. Using the same procedure as in Subsection 2.2, the estimate $\widehat{\boldsymbol{\gamma}}_r^{(b)}$ can be obtained, and then the low-dimensional projection can be defined as $\widehat{\boldsymbol{z}}_r^{(b)} := \boldsymbol{x}_r^{(b)} - \boldsymbol{X}_{-r}^{(b)}\widehat{\boldsymbol{\gamma}}_r^{(b)}$, which is used in acquiring the online debiased lasso estimator in Subsection 2.4. It is important to note that the size of

$\boldsymbol{R}^{(b)}$ and $\boldsymbol{T}^{(b)}$ are fixed, and they do not increase as more data batches arrive, which is vital to online or incremental learning.

2.4 Online debiased lasso estimator

Upon the arrival of the data batch $\mathcal{D}_b$, according to Han et al.[1], the ODL estimator for $\beta_{0,p},\ r = 1, ..., p$, is given by

$$\widehat{\beta}_{\text{on},r}^{(b)} := \widehat{\beta}_r^{(b)} + \left\{ \sum_{j=1}^{b} (\widehat{\boldsymbol{z}}_r^{(j)})^\top \boldsymbol{x}_r^{(j)} \right\}^{-1} \left\{ \sum_{j=1}^{b} (\widehat{\boldsymbol{z}}_r^{(j)})^\top \boldsymbol{y}^{(j)} - \sum_{j=1}^{b} (\widehat{\boldsymbol{z}}_r^{(j)})^\top \boldsymbol{X}^{(j)} \widehat{\boldsymbol{\beta}}^{(b)} \right\}. \tag{11}$$

However, the above formula still contains the historical data. To address this issue when computing $\widehat{\beta}_{\text{on},r}^{(b)}$, Han & Luo[1] uses the following statistics to utilize the information in the data instead of storing the entire dataset. These statistics are defined as

$$a_1^{(b)} := \sum_{j=1}^{b} (\widehat{\boldsymbol{z}}_r^{(j)})^\top \boldsymbol{x}_r^{(j)},\ a_2^{(b)} := \sum_{j=1}^{b} (\widehat{\boldsymbol{z}}_r^{(j)})^\top \boldsymbol{y}^{(j)},\ \boldsymbol{A}_1^{b} := \sum_{j=1}^{b} (\widehat{\boldsymbol{z}}_r^{(j)})^\top \boldsymbol{X}^{(j)} \widehat{\boldsymbol{\beta}}^{(b)},$$

where the updating rule for new data batch arriving is given by:

$$a_1^{(b)} = a_1^{(b-1)} + (\widehat{\boldsymbol{z}}_r^{(b)})^\top \boldsymbol{x}_r^{(b)},\ a_2^{(b)} = a_2^{(b-1)} + (\widehat{\boldsymbol{z}}_r^{(b)})^\top \boldsymbol{y}^{(b)},\ \boldsymbol{A}_1^{(b)} = \boldsymbol{A}_1^{(b-1)} + (\widehat{\boldsymbol{z}}_r^{(b)})^\top \boldsymbol{X}^{(b)} \widehat{\boldsymbol{\beta}}^{(b)}.$$

Similar to the statistics in Subsections 2.2 and 2.3, these three statistics remain in the same dimension when being updated. It is worth noting that during the parameter update process when new data arrives, these intermediate parameters are updated without increasing the dimensionality, that is, without increasing the storage requirements, which is a basic necessary condition for ODL and other online learning algorithms. It is one of the basic requirements in online or incremental learning.

Note that the ODL estimator $\widehat{\beta}_{\text{on},r}^{(b)}$ in (11) consists of the addition of two terms. The former is the r-th component of $\widehat{\boldsymbol{\beta}}_r^{(b)}$ in (5), which can be obtained by applying the online lasso method as discussed in Subsection 2.2. The latter is an expression combined by the three statistics and depends on the estimate $\widehat{\boldsymbol{z}}_r^{(b)}$, which can be obtained by applying the low-dimensional projection as discussed in Subsection 2.3.

Besides, according to Han et al.[1] and Zhang et al.[2], the standard error estimate of $\widehat{\beta}_{\text{on},r}^{(b)}$ can be calculated as $\widehat{\sigma}_\epsilon^{(b)}\widehat{\tau}_r^{(b)}$, where the value of $(\widehat{\sigma}_\epsilon^2)^{(b)}$ can be found using equation (9) in Subsection 2.2, and $\widehat{\tau}_r^{(b)}$ is determined by taking the square root of the sum of the squared values of $\widehat{\boldsymbol{z}}_r^{(j)}$ over j values from 1 to b, divided by the sum of the dot products between $\widehat{\boldsymbol{z}}_r^{(j)}$ and $\boldsymbol{x}_r^{(j)}$ over the same range. That is, $\widehat{\tau}_r^{(b)} = \sqrt{\sum_{j=1}^{b}(\widehat{\boldsymbol{z}}_r^{(j)})^\top \widehat{\boldsymbol{z}}_r^{(j)}} \big/ \sum_{j=1}^{b}(\widehat{\boldsymbol{z}}_r^{(j)})^\top \boldsymbol{x}_r^{(j)}$.

2.5 Tuning parameter selection

When performing cross-validation in an offline setting to select the tuning parameter λ, the entire dataset is typically split into training and testing sets. However, this method cannot be used in an online setting where the complete dataset is not available. To solve this problem, Han et al.[1] proposes a sample-splitting technique inspired by time series forecasting.

In this approach, the data is split into sequential batches. At each time point b, the training set consists of all the data batches up to time point $b-1$, denoted by $\mathcal{D}_1, ..., \mathcal{D}_{b-1}$. The current data batch $\mathcal{D}_b$ is then used as the testing set. This approach is sometimes referred to as $rolling\text{-}original\text{-}recalibration$, a technique that was first introduced by Tashman[8].

To begin with, Han et al.[1] estimates $\widehat{\boldsymbol{\beta}}^{(1)}$ using the first data batch $\mathcal{D}_1$ and select the tuning parameter λ_1 using classical offline cross-validation. Then, when the b-th data batch $\mathcal{D}_b$ arrives, Han et al.[1] calculates the prediction error $PE_b(\lambda)$ for each $\lambda \in T_\lambda$, where T_λ is a set of candidate values for λ, and define the optimal tuning parameter λ_b as the value that minimizes the prediction error $PE_b(\lambda)$. Specifically, the expressions are given by

$$PE_b(\lambda) = \frac{1}{n_b}||\boldsymbol{y}^{(b)} - \boldsymbol{X}^{(b)}\widehat{\boldsymbol{\beta}}^{(b-1)}(\lambda)||_2^2,\ \lambda \in T_\lambda \ \text{ and } \ \lambda_b := \underset{\lambda \in T_\lambda}{\arg\min}\, PE_b(\lambda).$$

The process of tuning parameter selection is shown in Figure 1. By using this method, Han et al.[1] gets the great property that one can adaptively select the optimal tuning parameter λ_b as new data batches arrive. Therefore, this can be used in the corresponding lasso estimator $\widehat{\boldsymbol{\beta}}^{(b)}(\lambda_b)$ as the starting point for ODL, which can greatly enhance the performance for ODL.

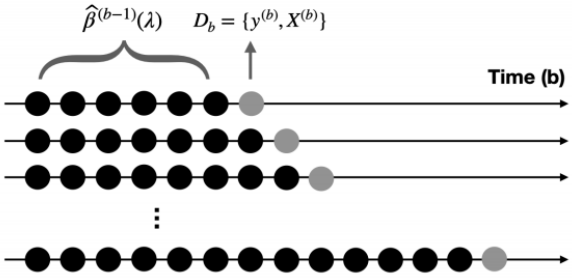

Figure 1 Flowchart of tuning parameter

3. Summary

In this section, I will summarize the process and framework for ODL based on Han et al.[1], and summarize the implementation and some details for ODL I encountered during the process of code writing.

3.1 Process and framework

According to Han et al.[1], the Online Debiased Lasso (ODL) procedure for statistical inference of $\beta_{0,r}$, with r ranging from 1 to p, can be summarized in a flowchart, as shown in Figure 2. The ODL procedure consists of two main blocks: the $online\ lasso\ estimation$ block and the $onlin\ low\text{-}dimensional\ projection$ block. Both blocks provide outputs that are used to calculate the online debiased lasso estimator and construct confidence intervals in real-time.

Upon the arrival of the new data batch $\mathcal{D}_b$, it is initially directed to the $online\ lasso\ estimation$ block, where the summary statistics $\boldsymbol{S}^{(b-1)}$ and $\boldsymbol{U}^{(b-1)}$ are updated to $\boldsymbol{S}^{(b)}$ and $\boldsymbol{U}^{(b)}$. This facilitates the updating of the lasso estimator $\widehat{\boldsymbol{\beta}}^{(b-1)}$ to $\widehat{\boldsymbol{\beta}}^{(b)}$ at specific grid values of the tuning parameters. Simultaneously, the dataset that generated the previous lasso estimate $\widehat{\boldsymbol{\beta}}^{(b-1)}$ is used as a training set, while the newly arrived $\mathcal{D}_b$ is utilized as a testing set to identify the tuning parameter λ_b that minimizes the prediction error as defined in Subsection 2.5. Then we obtain $\widehat{\boldsymbol{\beta}}^{(b)}(\lambda_b)$.

The selected λ_b is then passed to the $online\ low\text{-}dimensional\ projection$ block to calculate $\widehat{\boldsymbol{\gamma}}_r^{(b)}(\lambda_b)$. The online updating process in this block is similar to that of the $online\ lasso$

estimation block, except that the relevant summary statistics are $\boldsymbol{R}^{(b)}$ and $\boldsymbol{T}^{(b)}$. The estimate of projection vector $\widehat{\boldsymbol{z}}_r^{(b)}$ is outputted from this block. Using $\widehat{\boldsymbol{z}}_r^{(b)}$ and the lasso estimator $\widehat{\boldsymbol{\beta}}^{(b)}(\lambda_b)$, we can now calculate the debiased lasso estimator $\widehat{\beta}_{\text{on},r}^{(b)}$ and the estimated standard error $\widehat{\text{SE}}(\widehat{\beta}_{\text{on},r}^{(b)})$. The overall ODL procedure is shown in Figure 2.

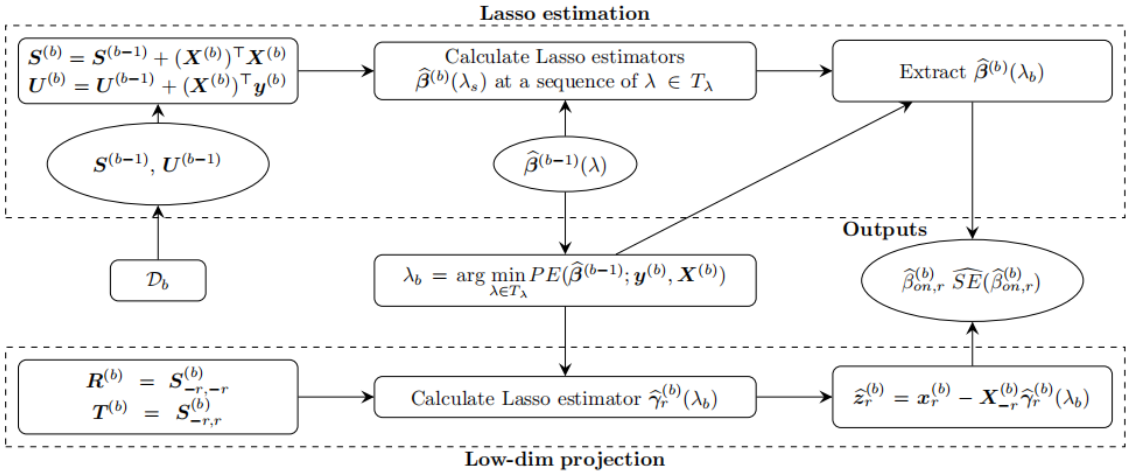

Figure 2 Flowchart of ODL

3.2 Implementation and details

The code implementation is mainly based on the flowchart of ODL in Figure 2. The key idea is that we should store the historical data using only some summary statistics and that we should use only these summary statistics and current data in the stream to calculate and incrementally update the goal parameters.

Initially, for $b = 1$, the offline debiased lasso actually consists of two parts, one is an offline lasso, and another is the debiasing procedure, which is implemented by the low-dimensional projection as discussed in Zhang et al.[2]. In fact, the low-dimensional projection is another lasso estimation problem with the coefficient $\boldsymbol{\beta}$ is substituted by $\boldsymbol{\gamma}_r$, $r = 1, ..., p$, which implies that the estimate $\widehat{\gamma}_r^{(1)}$ can be obtained similar to the procedure when estimating $\boldsymbol{\beta}^{(1)}$. Next, we should obtain the estimate $\widehat{\boldsymbol{\beta}}^{(1)}(\lambda)$ for $\lambda \in T_\lambda$, where T_λ are values at some grids. Here, we select λ via cross-validation and store the estimate values for $\lambda \in T_\lambda$ for parameter selection in the next round.

As mentioned in Subsection 2.2, Han et al.[1] does not specify how to estimate σ_ϵ^2 for

$b = 1$. I contacted the author (Ruijian Han) to ask for further details, but he didn't make it clear what method they used at this step. Therefore, to address this issue, I used the cross-validation method to obtain the estimator of $\widehat{\boldsymbol{\beta}}^{(1)}(\lambda)$. Cross-validation is not the most common approach for estimating the standard error in lasso and the bootstrap method is used in practice, however, the bootstrap method is computationally intensive, which is not suitable in an online setting.

For this reason, my code does not exactly reproduce the ODL algorithm as Han et al.[1] proposed, but it is only different from the ODL algorithm they proposed in the time point $b = 1$ of standard error estimation. The results in Subsection 4.2 show that the standard error estimation method of the ODL algorithm in Han et al.[1] is a little better than mine at this step, but there is not much difference, especially when the flow of streaming data becomes long, that is, as b increases. Except for this, I completed all the code reproduction based on the ODL algorithm and further found the simulation and code problem in Han et al.[1], where the detailed discussion is given in Subsection 4.2.

When the new data batch arrives, we update $\boldsymbol{S}^{(b-1)}$, $\boldsymbol{U}^{(b-1)}$, $\boldsymbol{R}^{(b-1)}$ and $\boldsymbol{T}^{(b-1)}$ by adding terms based on operations on current data batch. Then we use the gradient descent instead of the coordinate descent, which is used in lasso estimation when $b = 1$, to solve the lasso estimation. Similarly, we should obtain different estimate values for $\lambda \in T_\lambda$. Then, unlike the cross-validation used in parameter selection when $b = 1$, we now use the method mentioned in Subsection 2.5 to select λ. Once λ is selected, we then calculate the lasso estimator $\widehat{\boldsymbol{\gamma}}_r^{(b)}$ and $\widehat{\boldsymbol{z}}_r^{(b)}$ in low-dimensional projection, where the procedure is exactly the same as the situation when $b = 1$.

Then, we can obtain the online debiased lasso estimator $\widehat{\beta}_{\text{on},r}^{(b)}$ by using (11) in Subsection 2.4, and obtain the estimator of $(\sigma_\epsilon^2)^{(b)}$ by using (9) in Subsection 2.2.

4. Simulation

4.1 Setup

Following Han et al.[1], let us generate N_b samples in b data batches, denoted by $\mathcal{D}_1, ..., \mathcal{D}_b$, from a linear model with Gaussian noise $\epsilon_i \overset{iid}{\sim} \mathcal{N}(0, \sigma_\epsilon^2)$ and a sparse parameter vector $\boldsymbol{\beta}_0 \in \mathbf{R}^p$ and consider the following model:

$$y_i^{(j)} = \boldsymbol{\beta}_0^\top \boldsymbol{x}_i^{(j)} + \epsilon_i^{(j)},\ i = 1, ..., n_j;\ j = 1, ..., b,$$

where $\boldsymbol{x}_i^{(j)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$.

Let s_0 be the number of non-zero components of $\boldsymbol{\beta}_0$. Then set $s_0/2$ coefficients to be 1 (strong signals), and another $s_0/2$ to be 0.01 (weak signals). Let η be the learning rate in gradient descent. The setting is given by:

$$N_b = 420,\ p = 400,\ b = 12,\ s_0 = 6,\ n_j = 35 \text{ for } j = 1, ..., 12; \quad \eta = 0.005.$$

For the setting, two types of covariance matrices are used:

$$\text{(a)}\ \boldsymbol{\Sigma} = \boldsymbol{I}_p; \qquad \text{(b)}\ \boldsymbol{\Sigma} = \{0.5^{|i-j|}\}_{i,j=1,...,p}.$$

Then, I evaluated the estimation and inference criteria using 2 metrics: averaged absolute bias in estimating $\boldsymbol{\beta}$ (A.bias), averaged estimated standard error (ASE). I compared the proposed online debiased lasso method with the OLS estimator as a benchmark and reported the results in Tables 1-4. The simulation is conducted over 200 replications. Besides, I compared the results with the results in Han et al.[1] and found some errors. Further details are discussed in Subsection 4.2.

4.2 Some issues

When comparing my results with the results in Han et al.[1], I found that the values of A.bias in their results were very small, basically one-tenth of my corresponding values (A.bias-1), which is an order of magnitude difference. However, the values of ASE are

basically the same. This phenomenon is very strange. If we look at the relationship between A.bias and ASE, we will find that the values of A.bias in their reported tables are too small to be consistent with the values of ASE. Also, I ran OLS rather than ODL repeatedly to check the results for the A.bias from the simulation, and each time the simulation results showed that the values of A.bias should not be as small as they reported.

Then, I contacted the author (Ruijian Han) and told him about this issue. In the process of our communication, I found that the way they calculate A.bias is not suitable. According to Ruijian Han, to compute such a bias, first, take the mean of the biases among 200 replications. After taking 200 times, take the average among the estimator of each dimension. Then, take the absolute value on the p-dim vector. Finally, take the average among the different positions of the resulting vector for $\beta = 0,\ 0.01$ and 1.

I calculated A.bias according to this method, and it did reach the order of one-tenth of the original one, but the final result turned out to be positive and negative. In fact, in this method, when averaging the results of 200 replications for the first time, instead of taking the absolute value of the biases, it is directly added, and the result is that there are positive and negative biases and will cancel each other out, and eventually cause A.bias to be very small. I also reported the results using this method in the following tables. Besides, after comparing their results and conducting multiple rounds of 200 replications simulations, I found that the value of A.bias calculated in this way is sometimes positive and sometimes negative and there is roughly 10 times difference between this calculation method and mine, which implies that the estimator is fluctuating around the true value, rather than just on one side of the true value.

4.3 Results

Excluding random factors, it can be seen from Table 1-4 that the results of ASE in my simulation are almost consistent with Han et al.[1]. As for A.bias, since their original calculation method was inappropriate, I corrected it and then calculated it, which resulted in no comparison here. Nevertheless, I also performed calculations using their original method,

and the results show that the original method of calculation will cause the values to fluctuate wildly around zero, although the order of magnitude is small.

In general, my simulation results are basically consistent with their simulation results, which shows that there is basically no problem with my code reproduction work.

The following are the simulation results.

4.3.1 Case 1: $\boldsymbol{\Sigma} = \boldsymbol{I}_p$

Both Table 1 and Table 2 have the same setup: $N_b = 420,\ p = 400,\ b = 12,\ s_0 = 6,\ \boldsymbol{\Sigma} = \boldsymbol{I}_p$. Tuning parameter λ is selected from $T_\lambda = \{0.15, 0.20, 0.25, 0.30\}$. Simulation results are summarized over 200 replications. λ in the table is the value with the highest frequency among 200 replications at each step.

Table 1 reports the results in Han et al.[1], while Table 2 reports the results in the simulation with my code. The results reported for A.bias-T are calculated by using the corrected way, while the original way for calculating A.bias is not suitable since the results are fluctuating.

Table 1 Han et al.'s simulation for Case 1

	$\beta_{0,r}$	OLS	online debiased lasso					
data batch index			2	4	6	8	10	12
λ			0.30	0.30	0.20	0.15	0.15	0.15
A.bias	0	0.013	0.008	0.005	0.004	0.004	0.003	0.003
	0.01	0.026	0.007	0.009	0.008	0.003	0.002	0.002
	1	0.016	0.152	0.022	0.009	0.005	0.002	0.001
ASE	0	0.223	0.119	0.091	0.074	0.063	0.056	0.051
	0.01	0.226	0.119	0.091	0.074	0.063	0.056	0.051
	1	0.222	0.118	0.091	0.074	0.063	0.056	0.051

Table 2 My simulation for Case 1

	$\beta_{0,r}$	OLS	online debiased lasso					
data batch index			2	4	6	8	10	12
λ			0.30	0.30	0.20	0.15	0.15	0.15
	0	-0.001	0.0001	0.0001	0.00001	0.0001	0.0001	0.0001
A.bias	0.01	0.005	-0.003	0.001	0.003	0.001	0.002	0.001
	1	0.003	-0.177	-0.012	-0.006	-0.001	0.003	0.002
	0	0.222	0.159	0.100	0.078	0.066	0.058	0.052
ASE	0.01	0.220	0.159	0.100	0.078	0.066	0.058	0.052
	1	0.221	0.159	0.100	0.078	0.066	0.058	0.052
	0	0.181	0.073	0.070	0.057	0.049	0.044	0.041
A.bias-T	0.01	0.173	0.077	0.069	0.058	0.051	0.047	0.043
	1	0.174	0.207	0.074	0.057	0.048	0.043	0.040

4.3.2 Case 2: $\mathbf{\Sigma} = \{0.5^{|i-j|}\}_{i,j=1,\ldots,p}$

Both Table 3 and Table 4 have the same setup: $N_b = 420,\ p = 400,\ b = 12,\ s_0 = 6,\ \{0.5^{|i-j|}\}_{i,j=1,\ldots,p}$. Tuning parameter λ is selected from $T_\lambda = \{0.15, 0.20, 0.25, 0.30\}$. Simulation results are summarized over 200 replications. λ in the table is the value with the highest frequency among 200 replications at each step.

Table 3 reports the results in Han et al.[1], while Table 4 reports the results in the simulation with my code. The results reported for A.bias-T are calculated by using the corrected way, while the original way for calculating A.bias is not suitable since the results are fluctuating.

Table 3 Han et al.'s simulation for Case 2

	$\beta_{0,r}$	OLS	online debiased lasso					
data batch index			2	4	6	8	10	12
λ			0.30	0.30	0.20	0.15	0.15	0.15
	0	0.018	0.011	0.009	0.006	0.005	0.005	0.004
A.bias	0.01	0.015	0.004	0.004	0.002	0.003	0.004	0.004
	1	0.019	0.179	0.048	0.022	0.007	0.004	0.004
	0	0.288	0.120	0.093	0.076	0.066	0.060	0.054
ASE	0.01	0.287	0.120	0.092	0.076	0.066	0.060	0.054
	1	0.287	0.121	0.093	0.076	0.066	0.060	0.054

Table 4 My simulation for Case 2

data batch index	$\beta_{0,r}$	OLS	online debiased lasso					
			2	4	6	8	10	12
λ			0.30	0.30	0.20	0.15	0.15	0.15
A.bias	0	-0.0004	-0.0006	-0.0002	0.0002	0.0003	-1.562	0.00006
	0.01	0.009	0.05	0.047	0.043	0.038	0.039	0.038
	1	0.004	-0.028	0.061	0.058	0.053	0.053	0.052
ASE	0	0.289	0.144	0.094	0.075	0.064	0.057	0.052
	0.01	0.287	0.145	0.094	0.075	0.064	0.057	0.052
	1	0.280	0.144	0.094	0.074	0.064	0.057	0.052
A.bias-T	0	0.232	0.073	0.069	0.057	0.050	0.045	0.041
	0.01	0.240	0.088	0.087	0.070	0.063	0.058	0.054
	1	0.220	0.112	0.100	0.084	0.073	0.071	0.068

5. Conclusion

During my work for reproducing the ODL algorithm, I studied Han et al.[1] in detail and found some formula errors in Han et al.[1]. Besides, in the process of code reproduction, I almost wholly achieved the effect of the original paper, which shows that I have completed the reproduction of the code consummately; in addition, I also found the data errors of the original paper when conducting simulations, and communicated with the author to identify and investigate the issue, then used the corrected calculation method to report the results.

Besides, until March 2023, the latest version of Han et al.[1] on arXiv was submitted in 2021. It is not ruled out that the authors will resubmit a new version of the article and change some places in the future, which will lead to some inconsistencies in this thesis.

Through the code reproduction of the ODL algorithm, I have a better understanding of streaming data and online learning, and basically know some characteristics and necessary conditions of online learning, compared with offline learning. Based on the principle of the ODL algorithm and my code reproduction work, I think that maybe other offline regression algorithms, such as ridge regression, logistic regression, and even non-parametric regression, can similarly deduce an online learning version for streaming data so that we can address more complicated data mining problems on streaming data efficiently.

References

[1] HAN R, LUO L, LIN Y, et al. Online Debiased Lasso for Streaming Data[Z]. 2021. arXiv: 2106.05925 [math.ST].

[2] ZHANG C H, ZHANG S S. Confidence intervals for low dimensional parameters in high dimensional linear models[J]. Journal of the Royal Statistical Society: Series B: Statistical Methodology, 2014: 217-242.

[3] Van de GEER S, BÜHLMANN P, RITOV Y, et al. On asymptotically optimal confidence regions and tests for high-dimensional models[J/OL]. The Annals of Statistics, 2014, 42(3). https://doi.org/10.1214%2F14-aos1221. DOI: 10.1214/14-aos1221.

[4] JAVANMARD A, MONTANARI A. Confidence intervals and hypothesis testing for high-dimensional regression[J]. The Journal of Machine Learning Research, 2014, 15(1): 2869-2909.

[5] FRIEDMAN J, HASTIE T, HÖFLING H, et al. Pathwise coordinate optimization [J/OL]. The Annals of Applied Statistics, 2007, 1(2). https://doi.org/10.1214%2F07-aoas131. DOI: 10.1214/07-aoas131.

[6] DAUBECHIES I, DEFRISE M, DE MOL C. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint[J]. Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences, 2004, 57(11): 1413-1457.

[7] DONOHO D L, JOHNSTONE I M. Ideal spatial adaptation by wavelet shrinkage[J]. biometrika, 1994, 81(3): 425-455.

[8] TASHMAN L J. Out-of-sample tests of forecasting accuracy: an analysis and review [J]. International journal of forecasting, 2000, 16(4): 437-450.

Appendix

Code

odl.py

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold
import math
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm

import warnings
from sklearn.exceptions import ConvergenceWarning

##warnings.filterwarnings("ignore", category=
    ConvergenceWarning)

##########################################
##########################################
##########################################

# Function for generating data
def generate_data(n_samples, p):

    ## case 1
    X = np.random.randn(n_samples, p)
    #y = 1 * X[:, 0] + 1 * X[:, 1] + 1 * X[:, 2] + 0.01
        * X[:, 3] + 0.01 * X[:, 4] + 0.01 * X[:, 5] +
       np.random.randn(n_samples)

    ## Case 2
    #X = np.zeros((n_samples, p))
    #mean = np.zeros(p)
    #cov = np.zeros((p, p))
```

```
30      #for i in range(p):
31      #    for j in range(p):
32      #        cov[i][j] = 0.5 ** abs(i - j)
33      #for i in range(n_samples):
34      #    X[i, :] = np.random.multivariate_normal(mean=
            mean,cov=cov)
35      #y = 1 * X[:, 0] + 1 * X[:, 1] + 1 * X[:, 2] + 0.01
            * X[:, 3] + 0.01 * X[:, 4] + 0.01 * X[:, 5] +
            np.random.randn(n_samples)
36      y = 1 * X[:, 0] + 1 * X[:, 1] + 1 * X[:, 2] + 1 * X
            [:, 3] + 1 * X[:, 4] + 1 * X[:, 5] + 1 * X[:, 6]
             + 1 * X[:, 7] + 1 * X[:, 8] + 1 * X[:, 9] + 1 *
             X[:, 10] + 0.01 * X[:, 11] + 0.01 * X[:, 12] +
            0.01 * X[:, 13] + 0.01 * X[:, 14] + 0.01 * X[:,
            15] + 0.01 * X[:, 16] + 0.01 * X[:, 17] + 0.01 *
             X[:, 18] + 0.01 * X[:, 19] + 0.01 * X[:, 20] +
            np.random.randn(n_samples)
37      return X, y
38
39
40
41
42  # Function for generating streaming data
43  def stream_matrix(matrix, b):
44
45      sub_matrices = np.array_split(matrix, b, axis=0)
46
47      for sub_matrix in sub_matrices:
48          yield sub_matrix
49
50
51
52  ### Following Zhang & Zhang, the debiased lasso for D_1
53  def my_debiased_lasso(X, y):
54
55      n, p = X.shape
```

```
56      lambda0 = math.sqrt(2 * math.log(p) / n)
57      model = sm.OLS(y, X)
58      results = model.fit()
59      beta_hat = results.params.reshape(-1, 1)
60      beta_hat = np.zeros(p).reshape(-1, 1)
61      for iter_count in range(1000):
62          temp = 0
63          if iter_count > 1:
64              temp = sigma2_hat
65          sigma2_hat = (np.linalg.norm(y.reshape(-1, 1) -
                X @ beta_hat) ** 2) / n
66          lambda_hat = math.sqrt(sigma2_hat) * lambda0
67          clf = Lasso(alpha=lambda_hat, max_iter=1000,
               selection='random', tol=1e-6)
68          clf.fit(X, y)
69          beta_hat = clf.coef_.reshape(-1, 1)
70          if abs(sigma2_hat - temp) < 1e-4:
71              break
72
73      return beta_hat, sigma2_hat
74
75
76
77  # Debiased lasso for D_1
78  def debiased_lasso(X, y, alpha):
79
80      n, p = X.shape
81      clf = Lasso(alpha=alpha, max_iter=1000, selection='
            cyclic', tol=1e-6)
82      clf.fit(X, y)
83      beta_lasso = clf.coef_.reshape(-1, 1)
84
85      gamma = []
86      beta_debiased = []
87      for r in range(0, p):
88          Xr = np.delete(X, r, axis=1)
```

```
89          xr = X[:, r:r+1]
90          clf.fit(Xr, xr)
91          temp = clf.coef_.reshape(-1, 1)
92          gamma.append(temp)
93          z = xr - Xr @ gamma[r]
94          a1 = z.T @ xr
95          a2 = z.T @ y
96          A1 = z.T @ X
97          beta_debiased.append(beta_lasso[r, 0] + (a2 -
                A1 @ beta_lasso) / a1)
98
99      return beta_debiased
100
101
102
103 # 4.2
104 # Lasso CV for D_1; return lambda_1   corresponding to
      (2) in the paper
105 def lasso_cv(X, y, lambda_values, n_folds=5):
106
107     n, p = X.shape
108     kf = KFold(n_splits=n_folds)
109     cv_scores = np.zeros(len(lambda_values))
110
111     for i, lambda_n in enumerate(lambda_values):
112         scores = []
113         clf = Lasso(alpha=lambda_n, max_iter=1000,
                selection='cyclic', tol=1e-6)
114         for train_index, val_index in kf.split(X):
115             X_train, X_val = X[train_index], X[
                    val_index]
116             y_train, y_val = y[train_index], y[
                    val_index]
117             clf.fit(X_train, y_train)
118             beta = clf.coef_.reshape(-1, 1)
119             y_pred = X_val @ beta
```

```
            score = np.mean((y_val.reshape(-1, 1) -
                y_pred) ** 2)
            scores.append(score)
        cv_scores[i] = np.mean(scores)

    best_lambda = lambda_values[np.argmin(cv_scores)]
    sigma2_lambda = cv_scores[np.argmin(cv_scores)]
    return best_lambda, sigma2_lambda



# Debiased lasso CV for D_1; return lambda_1, sigma
    ^2^(1)
def debiased_lasso_cv(X, y, lambda_values, n_folds=5):

    n, p = X.shape
    kf = KFold(n_splits=n_folds)
    cv_scores = np.zeros(len(lambda_values))

    for i, lambda_n in enumerate(lambda_values):
        scores = []
        for train_index, val_index in kf.split(X):
            X_train, X_val = X[train_index], X[
                val_index]
            y_train, y_val = y[train_index], y[
                val_index]
            beta = debiased_lasso(X_train, y_train,
                lambda_n)
            beta = np.array(beta).reshape(p, 1)
            y_pred = X_val @ beta
            score = np.mean((y_val.reshape(-1, 1) -
                y_pred) ** 2)
            scores.append(score)
        cv_scores[i] = np.mean(scores)

    best_lambda = lambda_values[np.argmin(cv_scores)]
```

```
150         sigma2_lambda = cv_scores[np.argmin(cv_scores)]
151         return best_lambda, sigma2_lambda
152 
153 
154 ############################################################
155 ############################################################
156 ########################framework###########################
157 ############################################################
158 ############################################################
159 
160 
161 (rep, p, s0) = (200, 400, 6)
162 bias_ols_mean = np.zeros(p)
163 std_ols_mean = np.zeros(p)
164 bias_odl_mean = np.zeros(p)
165 std_odl_mean = np.zeros(p)
166 bias_odl_2_mean = np.zeros(p)
167 std_odl_2_mean = np.zeros(p)
168 bias_odl_4_mean = np.zeros(p)
169 std_odl_4_mean = np.zeros(p)
170 bias_odl_6_mean = np.zeros(p)
171 std_odl_6_mean = np.zeros(p)
172 bias_odl_8_mean = np.zeros(p)
173 std_odl_8_mean = np.zeros(p)
174 bias_odl_10_mean = np.zeros(p)
175 std_odl_10_mean = np.zeros(p)
176 lambda_all = np.zeros(p)
177 for i in range(rep):
178 
179     # Generate streaming data
180     #np.random.seed(1234)
181     (n_samples, p, b) = (420, 100, 12)
182     X, y = generate_data(n_samples, p)
183     D = np.concatenate((X, y.reshape(-1, 1)), axis=1)
184     stream = stream_matrix(D, b)
185 
```

```
186     # Perform OLS
187     model = sm.OLS(y, X)
188     results = model.fit()
189     beta_true = np.concatenate([np.array([1, 1, 1,
            0.01, 0.01, 0.01]), np.zeros(p - s0)])
190     #beta_true = np.concatenate([np.array([1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 0.01, 0.01, 0.01, 0.01, 0.01,
            0.01, 0.01, 0.01, 0.01, 0.01]), np.zeros(p - s0)
            ])
191     bias_ols = abs(results.params - beta_true)
192     std_ols = results.bse
193
194
195
196     # value of beta^(1), gamma_r^(1)
197     n1 = int(n_samples / b)
198     X1 = X[0:n1, ]
199     y1 = y[0:n1, ]
200     T_lambda = [0.15, 0.20, 0.25, 0.30]
201     lambda1, sigma2 = lasso_cv(X1, y1, T_lambda)
202     print("lambda1", lambda1)
203     clf = Lasso(alpha=lambda1, max_iter=1000, selection
            ='cyclic', tol=1e-6)
204     clf.fit(X1, y1)
205     beta_lasso = clf.coef_.reshape(-1, 1)
206     gamma1, beta1 = [], []
207     for r in range(0, p):
208         Xr = np.delete(X1, r, axis=1)
209         xr = X1[:, r:r+1]
210         clf.fit(Xr, xr)
211         temp = clf.coef_.reshape(-1, 1)
212         gamma1.append(temp)
213         z = xr - Xr @ gamma1[r]
214         a1 = z.T @ xr
215         a2 = z.T @ y1
216         A1 = z.T @ X1
```

```
217         beta1.append(beta_lasso[r, 0] + (a2 - A1 @
                beta_lasso) / a1)
218
219     beta1 = np.array(beta1).reshape(p, 1) ## \beta_{off
            }^{(1)}
220     gammas = list(gamma1)  ## \gamma_r^{(1)}
221     print("=======================================")
222
223
224     # beta^(1) at lambda grids
225     beta1lambda = []
226     for lam in T_lambda:
227         bbeta1 = []
228         for r in range(0, p):
229             Xr = np.delete(X1, r, axis=1)
230             xr = X1[:, r:r+1]
231             clf.fit(Xr, xr)
232             temp = clf.coef_.reshape(-1, 1)
233             gamma1.append(temp)
234             z = xr - Xr @ gamma1[r]
235             a1 = z.T @ xr
236             a2 = z.T @ y1
237             A1 = z.T @ X1
238             bbeta1.append(beta_lasso[r, 0] + (a2 - A1 @
                    beta_lasso) / a1)
239
240         bbeta1 = np.array(bbeta1).reshape(p, 1)
241         beta1lambda.append(bbeta1)
242
243
244
245     # Online framework
246     #learning_rate = 0.005
247     (j, N) = (0, 0)
248     S = np.zeros((p, p))
249     U = np.zeros((p, 1))
```

```
250     lambda_choose = []
251     Rs, Ts, a1s, a2s, A1s, b1s = [], [], [], [], [], []
252     beta_lasts, betas, beta_on, tau, std = [], [], [],
            [], []
253     for sub_matrix in stream:
254
255         # 2.1 Online lasso
256         j = j + 1
257         N = N + sub_matrix.shape[0]
258         X = sub_matrix[:, :p]
259         y = sub_matrix[:, p:p+1]
260         S = S + X.T @ X
261         U = U + X.T @ y
262         ## j = 1 (offline)
263         if j == 1:
264             beta = beta1  ## \beta_{off}^{(1)}
265             betas = list(beta1lambda) ## \beta_{off
                    }^{(1)} at lambda grids
266             print("sigma2 = ", sigma2)
267         ## j > 1 (online)
268         else:
269             beta_lasts = list(betas)
270             betas.clear()
271             for lambdab in T_lambda:
272                 bbeta = np.zeros(p).reshape(-1, 1)
273                 for iter_count in range(1000):
274                     grad = (S @ bbeta - U) / N
275                     bbeta = bbeta - learning_rate *
                            grad
276                     bbeta = np.sign(bbeta) * np.maximum
                            (np.abs(bbeta) - learning_rate *
                             lambdab, 0)
277                     if np.linalg.norm(grad, ord='fro')
                            < 1e-6:
278                         break
279                 betas.append(bbeta)
```

```
280
281
282
283             # 2.4 Tuning parameter selection
284             val = []
285             for mybeta in beta_lasts:
286                 temp = np.linalg.norm(y - X @ mybeta)
                        ** 2
287                 val.append(temp)
288             index = val.index(min(val))
289             beta = betas[index]
290             lambda_choose.append(T_lambda[index])
291             print("choose lambda = ", T_lambda[index])
292
293
294
295             ## estimator of sigma^2^(b)    (9) in the
                    paper
296             rr = np.linalg.norm(y - X @ beta) ** 2
297             sigma2 = (N - sub_matrix.shape[0]) * sigma2
                    / N + rr / N
298             print("sigma2 = ", sigma2)
299
300
301
302         # 2.2 Online low-dimensional projection
303         tau.clear()
304         std.clear()
305         for r in range(0, p):
306             Xr = np.delete(X, r, axis=1)
307             xr = X[:, r:r+1]
308             R = np.zeros((p-1, p-1))
309             T = np.zeros((p-1, 1))
310
311             ## step 1
312             if j == 1:
```

```
313                 R = Xr.T @ Xr
314                 T = Xr.T @ xr
315                 Rs.append(R)
316                 Ts.append(T)
317             else:
318                 Rs[r] = Rs[r] + Xr.T @ Xr
319                 Ts[r] = Ts[r] + Xr.T @ xr
320                 ggamma = np.zeros(p-1).reshape(-1, 1)
321                 for iter_count in range(1000):
322                     grad = (Rs[r] @ ggamma - Ts[r]) / N
323                     ggamma = ggamma - learning_rate *
                            grad
324                     ggamma = np.sign(ggamma) * np.
                            maximum(np.abs(ggamma) -
                            learning_rate * lambdab, 0)
325                     if np.linalg.norm(grad, ord='fro')
                            < 1e-6:
326                         break
327                 gammas[r] = ggamma
328
329             ## z_r^(j)
330             zr = xr - Xr @ gammas[r]
331
332
333
334         # 2.3 Online debiased lasso estimator
335             ## a_1^(b), a_2^(b), A_1^(b)
336             if j == 1:
337                 a1s.append(zr.T @ xr)
338                 a2s.append(zr.T @ y)
339                 A1s.append(zr.T @ X)
340                 b1s.append(zr.T @ zr)
341             else:
342                 a1s[r] = a1s[r] + zr.T @ xr
343                 a2s[r] = a2s[r] + zr.T @ y
344                 A1s[r] = A1s[r] + zr.T @ X
```

```
345                 b1s[r] = b1s[r] + zr.T @ zr
346
347             ## beta_{on,r}^(b)
348             if j == 1:
349                 beta_on.append(beta[r,0] + (a2s[r] -
                        A1s[r] @ beta) / a1s[r])
350             else:
351                 beta_on[r] = beta[r,0] + (a2s[r] - A1s[
                        r] @ beta) / a1s[r]
352
353             ## value of tau_r^(b)
354             tau.append(math.sqrt(b1s[r]) / a1s[r])
355
356             ## estimated standard error
357             std.append(math.sqrt(sigma2) * tau[r])
358
359
360
361         ## save the vale when b = 2, 4, 6, 8, 10
362         if j == 2:
363             beta_odl_2 = np.zeros(p)
364             std_odl_2 = np.zeros(p)
365             for r in range(0, p):
366                 beta_odl_2[r] = np.array(beta_on)[r]
367                 std_odl_2[r] = np.array(std)[r]
368             bias_odl_2 = abs(beta_odl_2 - beta_true)
369         elif j == 4:
370             beta_odl_4 = np.zeros(p)
371             std_odl_4 = np.zeros(p)
372             for r in range(0, p):
373                 beta_odl_4[r] = np.array(beta_on)[r]
374                 std_odl_4[r] = np.array(std)[r]
375             bias_odl_4 = abs(beta_odl_4 - beta_true)
376         elif j == 6:
377             beta_odl_6 = np.zeros(p)
378             std_odl_6 = np.zeros(p)
```

```
            for r in range(0, p):
                beta_odl_6[r] = np.array(beta_on)[r]
                std_odl_6[r] = np.array(std)[r]
            bias_odl_6 = abs(beta_odl_6 - beta_true)
        elif j == 8:
            beta_odl_8 = np.zeros(p)
            std_odl_8 = np.zeros(p)
            for r in range(0, p):
                beta_odl_8[r] = np.array(beta_on)[r]
                std_odl_8[r] = np.array(std)[r]
            bias_odl_8 = abs(beta_odl_8 - beta_true)
        elif j == 10:
            beta_odl_10 = np.zeros(p)
            std_odl_10 = np.zeros(p)
            for r in range(0, p):
                beta_odl_10[r] = np.array(beta_on)[r]
                std_odl_10[r] = np.array(std)[r]
            bias_odl_10 = abs(beta_odl_10 - beta_true)

        ## print b
        print("j=", j)



    # bias ODL & std ODL over the streaming
    beta_odl = np.zeros(p)
    std_odl = np.zeros(p)
    for r in range(0, p):
        beta_odl[r] = np.array(beta_on)[r]
        std_odl[r] = np.array(std)[r]
    bias_odl = abs(beta_odl - beta_true)



    # summary over the replications
    bias_ols_mean += bias_ols
```

```
415         std_ols_mean += std_ols
416         bias_odl_mean += bias_odl
417         std_odl_mean += std_odl
418         bias_odl_2_mean += bias_odl_2
419         std_odl_2_mean += std_odl_2
420         bias_odl_4_mean += bias_odl_4
421         std_odl_4_mean += std_odl_4
422         bias_odl_6_mean += bias_odl_6
423         std_odl_6_mean += std_odl_6
424         bias_odl_8_mean += bias_odl_8
425         std_odl_8_mean += std_odl_8
426         bias_odl_10_mean += bias_odl_10
427         std_odl_10_mean += std_odl_10
428 
429 
430 
431 ############################################################
432 ############################################################
433 ##################Output the results########################
434 ############################################################
435 ############################################################
436 
437 
438 print("+++++++++++++++The Results+++++++++++++++")
439 
440 bias_ols_mean = bias_ols_mean / rep
441 std_ols_mean = std_ols_mean / rep
442 bias_odl_2_mean = bias_odl_2_mean / rep
443 std_odl_2_mean = std_odl_2_mean / rep
444 bias_odl_4_mean = bias_odl_4_mean / rep
445 std_odl_4_mean = std_odl_4_mean / rep
446 bias_odl_6_mean = bias_odl_6_mean / rep
447 std_odl_6_mean = std_odl_6_mean / rep
448 bias_odl_8_mean = bias_odl_8_mean / rep
449 std_odl_8_mean = std_odl_8_mean / rep
450 bias_odl_10_mean = bias_odl_10_mean / rep
```

```
451 std_odl_10_mean = std_odl_10_mean / rep
452 bias_odl_mean = bias_odl_mean / rep
453 std_odl_mean = std_odl_mean / rep
454
455 # results for bais and std for 1, 0.01, 0 for b=2, 4,
      6, 8, 10, 12
456 s = int(s0/2)
457 bias_ols_1 = np.mean(bias_ols_mean[:s])
458 bias_ols_2 = np.mean(bias_ols_mean[s:s0])
459 bias_ols_3 = np.mean(bias_ols_mean[s0:])
460 std_ols_1 = np.mean(std_ols_mean[:s])
461 std_ols_2 = np.mean(std_ols_mean[s:s0])
462 std_ols_3 = np.mean(std_ols_mean[s0:])
463 bias_odl_2_1 = np.mean(bias_odl_2_mean[:s])
464 bias_odl_2_2 = np.mean(bias_odl_2_mean[s:s0])
465 bias_odl_2_3 = np.mean(bias_odl_2_mean[s0:])
466 std_odl_2_1 = np.mean(std_odl_2_mean[:s])
467 std_odl_2_2 = np.mean(std_odl_2_mean[s:s0])
468 std_odl_2_3 = np.mean(std_odl_2_mean[s0:])
469 bias_odl_4_1 = np.mean(bias_odl_4_mean[:s])
470 bias_odl_4_2 = np.mean(bias_odl_4_mean[s:s0])
471 bias_odl_4_3 = np.mean(bias_odl_4_mean[s0:])
472 std_odl_4_1 = np.mean(std_odl_4_mean[:s])
473 std_odl_4_2 = np.mean(std_odl_4_mean[s:s0])
474 std_odl_4_3 = np.mean(std_odl_4_mean[s0:])
475 bias_odl_6_1 = np.mean(bias_odl_6_mean[:s])
476 bias_odl_6_2 = np.mean(bias_odl_6_mean[s:s0])
477 bias_odl_6_3 = np.mean(bias_odl_6_mean[s0:])
478 std_odl_6_1 = np.mean(std_odl_6_mean[:s])
479 std_odl_6_2 = np.mean(std_odl_6_mean[s:s0])
480 std_odl_6_3 = np.mean(std_odl_6_mean[s0:])
481 bias_odl_8_1 = np.mean(bias_odl_8_mean[:s])
482 bias_odl_8_2 = np.mean(bias_odl_8_mean[s:s0])
483 bias_odl_8_3 = np.mean(bias_odl_8_mean[s0:])
484 std_odl_8_1 = np.mean(std_odl_8_mean[:s])
485 std_odl_8_2 = np.mean(std_odl_8_mean[s:s0])
```

```
std_odl_8_3 = np.mean(std_odl_8_mean[s0:])
bias_odl_10_1 = np.mean(bias_odl_10_mean[:s])
bias_odl_10_2 = np.mean(bias_odl_10_mean[s:s0])
bias_odl_10_3 = np.mean(bias_odl_10_mean[s0:])
std_odl_10_1 = np.mean(std_odl_10_mean[:s])
std_odl_10_2 = np.mean(std_odl_10_mean[s:s0])
std_odl_10_3 = np.mean(std_odl_10_mean[s0:])
bias_odl_1 = np.mean(bias_odl_mean[:s])
bias_odl_2 = np.mean(bias_odl_mean[s:s0])
bias_odl_3 = np.mean(bias_odl_mean[s0:])
std_odl_1 = np.mean(std_odl_mean[:s])
std_odl_2 = np.mean(std_odl_mean[s:s0])
std_odl_3 = np.mean(std_odl_mean[s0:])


print("bias OLS 1: ", bias_ols_1)
print("bias OLS 0.01: ", bias_ols_2)
print("bias OLS 0: ", bias_ols_3)
print("std OLS 1: ", std_ols_1)
print("std OLS 0.01: ", std_ols_2)
print("std OLS 0: ", std_ols_3)
print("-----------------------------------")
print("b=2, bias ODL 1: ", bias_odl_2_1)
print("b=2, bias ODL 0.01: ", bias_odl_2_2)
print("b=2, bias ODL 0: ", bias_odl_2_3)
print("b=2, std ODL 1: ", std_odl_2_1)
print("b=2, std ODL 0.01: ", std_odl_2_2)
print("b=2, std ODL 0: ", std_odl_2_3)
print("-----------------------------------")
print("b=4, bias ODL 1: ", bias_odl_4_1)
print("b=4, bias ODL 0.01: ", bias_odl_4_2)
print("b=4, bias ODL 0: ", bias_odl_4_3)
print("b=4, std ODL 1: ", std_odl_4_1)
print("b=4, std ODL 0.01: ", std_odl_4_2)
print("b=4, std ODL 0: ", std_odl_4_3)
print("-----------------------------------")
```

```
print("b=6, bias ODL 1: ", bias_odl_6_1)
print("b=6, bias ODL 0.01: ", bias_odl_6_2)
print("b=6, bias ODL 0: ", bias_odl_6_3)
print("b=6, std ODL 1: ", std_odl_6_1)
print("b=6, std ODL 0.01: ", std_odl_6_2)
print("b=6, std ODL 0: ", std_odl_6_3)
print("-----------------------------------")
print("b=8, bias ODL 1: ", bias_odl_8_1)
print("b=8, bias ODL 0.01: ", bias_odl_8_2)
print("b=8, bias ODL 0: ", bias_odl_8_3)
print("b=8, std ODL 1: ", std_odl_8_1)
print("b=8, std ODL 0.01: ", std_odl_8_2)
print("b=8, std ODL 0: ", std_odl_8_3)
print("-----------------------------------")
print("b=10, bias ODL 1: ", bias_odl_10_1)
print("b=10, bias ODL 0.01: ", bias_odl_10_2)
print("b=10, bias ODL 0: ", bias_odl_10_3)
print("b=10, std ODL 1: ", std_odl_10_1)
print("b=10, std ODL 0.01: ", std_odl_10_2)
print("b=10, std ODL 0: ", std_odl_10_3)
print("-----------------------------------")
print("bias ODL 1: ", bias_odl_1)
print("bias ODL 0.01: ", bias_odl_2)
print("bias ODL 0: ", bias_odl_3)
print("std ODL 1: ", std_odl_1)
print("std ODL 0.01: ", std_odl_2)
print("std ODL 0: ", std_odl_3)
```