International Journal of Data Science and Analytics (2025) 20:5017–5032
https://doi.org/10.1007/s41060-025-00766-3

REGULAR PAPER

Linear adaptive filtering for regression in data streams

Jorge Igual[1] · Heitor Murilo Gomes[2] · Bernhard Pfahringer[3,4] · Albert Bifet[4,5]

Received: 16 August 2024 / Accepted: 15 March 2025 / Published online: 6 May 2025
© The Author(s) 2025

Abstract
Many applications in supervised learning of evolving data streams deal with regression, since they need to forecast numeric values in a fast and accurate way. Most of the current regression methods are inspired by the state-of-the-art data stream classifiers adapted to the regression problem. In this paper, we present a novel approach to the problem based on classical linear adaptive filtering theory. The linearity allows to obtain simple general models highly accurate without needing to use ensemble learners. We study the recursive version of these methods as they are the ones that satisfy the stream requirements and we include in their formulation the ability to track the drift of the data in stream scenarios. We show how they integrate in an easy and fast way the adaptation to changing environments avoiding the use of explicit drift detectors. We apply these methods to classical datasets in stream regression and show how they can outperform them in drifting cases when the linear model is a good approximation to the problem.

Keywords Regression · Data streams · Linear filters · Adaptive filtering · Least mean squares · Recursive least squares · Ensemble learners · Regression trees

1 Introduction

Continuously building and updating predictive models from data streams is a challenging task motivated by the ever-growing number of real-time data sources. Supervised learning has received significant attention from the research community, leading to a multitude of algorithms designed to address the many challenges inherent to building models with an endless influx of data [12]. These challenges include updating the model given restrictive computational resources, coping with changes to the underlying data distribution and maintaining predictive accuracy over time. While substantial progress has been made, applying existing methodologies to data stream regression poses unique challenges. Unlike classification, regression involves a continuous target variable, making it more sensitive to subtle shifts in the underlying data distribution. For example, concept drift in regression outputs can manifest as gradual changes in trends or abrupt shifts in the range of target values, requiring methods that can effectively distinguish between noise and meaningful drift. Additionally, balancing model adaptability with stability becomes more intricate, as overfitting to transient patterns can severely impact the accuracy of predictions, particularly in domains with high variability or sparse target updates.

Distribution changes in the target variable, often referred to as real concept drifts [8], are pervasive in a data stream supervised task. There are several approaches to address such problem, mainly proactive and reactive approaches. Reactive approaches, which often incorporate a drift detection algorithm alongside the learning model, have gained a lot of attention recently, specially for classification problems [10]. However, proactive strategies, where the learning model is

✉ Heitor Murilo Gomes
heitor.gomes@vuw.ac.nz

Jorge Igual
jigual@dcom.upv.es

Bernhard Pfahringer
bernhard@waikato.ac.nz

Albert Bifet
albert.bifet@waikato.ac.nz

1 Instituto de Telecomunicaciones y Aplicaciones Multimedia (ITEAM), Departamento de Comunicaciones, Universitat Politècnica de València, 46022 Valencia, Spain

2 School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand

3 Computer Science Department, University of Waikato, Waikato, New Zealand

4 University of Waikato, Hamilton, New Zealand

5 LTCI, Telecom Paris, IP Paris, Paris, France

constantly reset using a tumbling or sliding window, are also effective and widely used [21].

In this work, we focus on using linear filtering for regression in streaming data. Linear filtering allows us to build efficient and accurate proactive general models without the need for ensemble learners or detection algorithms (as in reactive approaches). Even though ensemble methods coupled with detection algorithms are popular for supervised learning in data streams, they can be somewhat inefficient in terms of the computational resources required, which is a key concern in most streaming scenarios. Adaptive linear filtering is a firmly established tool in signal processing [13], originated in the classical theory of linear systems and very attractive in terms of computational resources used. Linear filters process time-varying input signals to produce output signals, subject to the constraint of linearity. They are naturally suited to be used in regression in real-time systems learning on large quantities of data.

There are already several works trying to leverage existing regression models from batch data to a streaming scenario, such as regression decision trees [16], rule-based models [1] or ensembles methods [9]. Our aim is to show that the classical adaptive linear filtering methodology can be used in regression of streaming data obtaining excellent performance in terms of error and resources used.

The main contributions of this paper include:

- Closing the gap between classical adaptive filters and data stream mining for regression tasks;
- Extending the theoretical linear regression solution based on normal equations, to a practical algorithm for data streams using iterative estimates of correlations;
- Improving adaptive filtering techniques to obtain better performance in terms of computational resources;
- To show how fast the filters can adapt to any drift in the data stream;
- An extensive empirical evaluation comparing adaptive filters with methods from the data stream regression literature.

To the best of our knowledge, this is the first paper to show the benefits of linear filtering for regression on evolving data streams.

The rest of this paper is organized as follows. Section 2 reviews the state of the art in data stream regression algorithms. Section 3 introduces the problem of data stream regression and how it can be stated from an adaptive linear filtering perspective. Sections 4 and 5 present the recursive least squares and the least mean squares approaches, respectively, and how the algorithms can be adapted to an evolving data stream. In Sect. 6, we present a new algorithm that speeds up traditional solutions. Section 7 shows the experimental evaluation results and corresponding discussions. Section 8 explains how the basic linear filtering model can be extended to include temporal dependencies, and finally, the paper concludes in Sect. 9.

2 Related work

To date, there are several algorithms proposed for data stream regression. These regression methods do not explicitly incorporate a traditional notion of time; instead, they implicitly account for the sequential nature of data arrival.

The **Adaptive Model Rules (AMRules)** [1] learns both an ordered and an unordered rule set from a data stream. To detect and adapt to concept drifts, each rule is associated with a Page–Hinkley drift detector [22], which prunes the rule set given changes in the incoming data.

The **Fast and Incremental Model Trees (FIMT-DD)** [16] is a successful regression decision tree adaptation to streaming data. Similarly to the Hoeffding Tree algorithm [5], FIMT-DD starts with an empty tree that keeps statistics from arriving data until a grace period is reached, and split decisions are made. The features are ranked according to their variance w.r.t the target variable to decide for splits, and if the two best-ranked differ by at least the Hoeffding Bound [14], the node splits. FIMT-DD includes a change detection scheme that periodically flags and adapts branches of the tree where the variance increases are significant. In [18], the authors extend FIMT-DD by proposing the **Online Regression Trees with Options (ORTO)** algorithm. ORTO introduces 'option' nodes, which allow an instance to follow all the branches available in a tree node. This approach is similar to the Hoeffding adaptive trees (HAT) [3] algorithm.

Ensemble learners are widely adopted for supervised data stream problems due to their effectiveness in handling evolving data distributions and concept drift. These learners offer improved predictive performance compared to single learners, on average. In dynamic streaming scenarios, where shifts in data patterns are anticipated, ensemble methods provide a strategic advantage. They facilitate selective resets, which involve retraining some of the base learners from scratch. The development of ensembles of regressors attracted less attention than ensembles of classifiers for streaming data, even though there is a sizeable amount of literature on this topic for batch learning [11]. For streaming data, Ikonomovska et al. [17] proposed the **online random forest (ORF)** and **online bagging (OBag)** ensembles that use the FIMT-DD as the base learner. Based on empirical experiments, the authors concluded that the ORTO-A (online option trees with averaging) outperformed both OBag and ORF in terms of mean squared error (MSE). The **Adaptive Random Forest regressor (ARF-Reg)** [9], an adaptation of the data stream classifier [10] of the same name, was proposed. ARF-Reg builds a forest of FIMT-DD trees as ORF, and the main dif-

ference between these algorithms is that ARF-Reg employs one instance of a drift detector (Adaptive WINdow (ADWIN) algorithm [2]) per tree, which allows the algorithm to selectively reset base learners when drifts are identified. More recently, Self-Optimizing K-Nearest Leaves (SOKNL) [26] has been introduced as a significant improvement upon ARF-Reg. It enhances the prediction process by generating a compressed representation of each leaf (centroid). During prediction, instead of involving all trees in the forest, SOKNL calculates the distances between the input instance and the centroids from relevant leaves. Only the k trees with the smallest distances are utilized for the prediction. Moreover, the algorithm introduces a mechanism for tuning the k values dynamically and automatically, based on historical information.

While existing methods for data stream regression provide valuable tools, they also exhibit notable limitations. Algorithms like FIMT-DD and its derivatives (e.g., ORTO) excel at handling large, high-velocity data streams but can struggle with model interpretability and computational efficiency in complex scenarios. Ensemble methods, such as ARF-Reg and SOKNL, improve predictive accuracy and resilience to concept drift by leveraging multiple base learners, yet they often come with increased computational costs, memory requirements and reliance on specific drift detectors, which may falter in noisy or highly dynamic data streams. To address these challenges, we propose an adaptive linear filtering framework that prioritizes computational efficiency, adaptability to data drift, and ease of implementation. This approach uses a parametric model requiring the estimation of only the filter coefficients, ensuring rapid responses to changes in data streams through continuous updates. In the remainder of the paper, we introduce this method, discuss its adaptation to streaming scenarios and conduct a comparative analysis with state-of-the-art approaches.

3 Linear regression and adaptive filtering

Regression analysis consists of obtaining a model that describes the relationship between the outcome of the interesting variables $\mathbf{y}(t) = [y_1(t), \ldots, y_M(t)]^T$ (the M responses) and some related variables (the N predictors or features) $\mathbf{x}(t) = [x_1(t), \ldots, x_N(t)]^T$.

$$\mathbf{y}(t) = f(\mathbf{x}(t)). \tag{1}$$

We will assume that t represents the *time* variable throughout the paper as usual, but t can be related to any variable indicating some order in the instances in that domain, e.g., spatial coordinates, wavelength, frequency, etc.

In this paper, we will assume the typical data stream scenario $M = 1$, e.g., only one response, and the linear case $f(\mathbf{x}(t)) = \mathbf{x}(t)$ in Eq. (1). The final model for the single outcome $y(t)$ reads, in matrix notation:

$$y(t) = \mathbf{c}^T(t)\mathbf{x}(t), \tag{2}$$

where $\mathbf{c}(t)$ is the coefficients vector to be estimated, $\mathbf{c}(t) = [c_1(t), \ldots, c_N(t)]^T$.

Equation (2) can be stated as a linear filtering problem: the way the input signal (the feature or attribute vector $\mathbf{x}(t)$) is processed by the filter $\mathbf{c}(t)$ in order to extract the useful information (the filter output $y(t)$). In addition, introducing the data streaming factor, it can be stated as an adaptive filtering problem, i.e., a time variant filter $\mathbf{c}(t)$ that is changing continuously in order to adapt to the unknown underlying changing nature of the data. By adjusting the filter coefficients dynamically, the filter learns the underlying linear relationships between attributes. The coefficients are adjusted to minimize the corresponding cost function, driving the filter to better capture the relationship between the inputs and the output. (Positive coefficients show a direct relationship while negative values show an inverse one; the magnitude of the coefficient represents the dependency strength.) The filter dynamically updates its coefficients, allowing it to track and model time-varying dependencies between attributes.

From an adaptive filtering point of view, the data stream model in Eq. (2) requires the determination of two different issues: the cost function that measures the goodness of the filter and the way that this is implemented in practice, i.e., how we obtain a practical algorithm that tries to optimize the filter behavior in an adaptive environment. (The input data is used only once.) The model in Eq. (2) can be easily extended to include a bias term by introducing a new term in the filter and increasing the size of the input vector $\mathbf{x}(t)$ to include a one value.

From a theoretical point of view, if the joint probability distribution $p(\mathbf{x}(t), y(t))$ was known, we could obtain the posterior distribution and the corresponding Bayesian optimal estimate, that is, the mean posterior expected value $E[y(t)|\mathbf{x}(t)]$. Other Bayesian estimates such as the maximum a posteriori MAP estimate also take into account the posterior distribution in order to make a decision; in this case, the MAP estimate is the one that maximizes the posterior probability $p(y(t)|\mathbf{x}(t))$. All these Bayesian point-wise solutions require the knowledge of the posterior probability. But this is never the case in real-world data streams. In a more realistic scenario, the problem is transformed in the minimization of some loss function. We compute the mean squared error between the model's predicted value $\hat{y}(t)$ and the true output value $y(t)$. This approach is equivalent to assuming a residual Gaussian distribution.

$$\varepsilon_{MSE}(t) = E[|y(t) - \hat{y}(t)|^2]. \tag{3}$$

The problem in Eq. (3) consists on the estimation of the $\mathbf{c}(t)$ vector that minimizes the mean squared error $\varepsilon_{MSE}(t)$. It is obtained by solving the normal equations.

$$\mathbf{R}(t)\mathbf{c}(t) = \mathbf{r}(t), \tag{4}$$

where $\mathbf{R}(t)$ is the correlation matrix of the input $\mathbf{R}(t) = E[\mathbf{x}(t)\mathbf{x}^T(t)]$ and $\mathbf{r}(t)$ is the cross-correlation between the input and output $\mathbf{r}(t) = E[\mathbf{x}(t)y(t)]$.

As we mentioned, in practice, the true distributions and corresponding second order statistics are unknown, so the correlations must be estimated numerically using the set of available pair instances $(\mathbf{x}(t), y(t)), t = 1, \ldots L$. In a block-wise or offline scenario, the correlations can be computed with a high accuracy if the number of samples L is large enough. The solution to Eq. (4) is:

$$\mathbf{c}(L) = \mathbf{R}^{-1}(L)\mathbf{r}(L), \tag{5}$$

where:

$$\mathbf{R}(L) = \frac{1}{L}\sum_{t=1}^{L}\mathbf{x}(t)\mathbf{x}^T(t), \tag{6}$$

$$\mathbf{r}(L) = \frac{1}{L}\sum_{t=1}^{L}\mathbf{x}(t)y(t). \tag{7}$$

3.1 Adaptive linear filtering for data streams

In a stationary environment, the optimal regressor vector $\mathbf{c}$ can be estimated once with Eq. (5) and be used for future predictions while the environment does not change. But the batched solution given by Eqs. (6) and (7) is not possible in data streams because it requires the reuse of previous instances and the size of the $\mathbf{x}(t)$ vector would increase continuously. One option is to use only the first L samples, but even in a stationary case this is a bad strategy since the larger the number of samples L used in the estimates, the better they are and the corresponding solution.

One way to use all the observations in the correlation estimates is to update the sample estimates using a recursive formula with a sliding window of size L; e.g., when a new sample arrives at $t = L+\tau$, the estimates are updated removing the contribution of feature $\mathbf{x}(\tau)$ and adding the new term due to $\mathbf{x}(L+\tau)$ in Eqs. (6) and (7). Obviously, there is a trade-off between the accuracy of the sample estimates and the sample size L as it happens with the computational cost. In data streams, the window size is $L = 1$, obtaining an iterative estimate of the correlations. This allows to track the changes in the statistics when the time stationarity of the features is not satisfied during a drifting period and to use all the samples only once, but also increases the variance of the estimates. If the probability distribution of the features does not change $p(\mathbf{x}(t_1)) = p(\mathbf{x}(t_2))$, we can keep all the previous samples in the recursion.

$$\mathbf{R}(t) = (1-\alpha)\mathbf{R}(t-1) + \alpha\mathbf{x}(t)\mathbf{x}^T(t), \tag{8}$$

$$\mathbf{r}(t) = (1-\alpha)\mathbf{r}(t-1) + \alpha\mathbf{x}(t)y(t), \tag{9}$$

with $\alpha = 1/t$ in order to average proportionally according to the length of the two terms in the recursion formulas, the correlation value at $t-1$ and the update term due to the new observation at t. Equations (8) and (9) can be easily modified in order to change the weights of each term in case we want not to give the same importance to all samples. We will explore this idea later in the paper.

Another problem with the classical linear regression solution in Eq. (5) from a data streams point of view is the computational cost, since it requires the inversion of the N-dimensional matrix $\mathbf{R}(t)$ (the computational complexity of the basic algorithm to invert a $D \times D$ matrix is $O(D^3)$). However, this can be avoided by applying the matrix inversion lemma [27], $(\mathbf{I}+\mathbf{A}\mathbf{B}^T)^{-1} = \mathbf{I} - \mathbf{A}(\mathbf{I}+\mathbf{B}^T\mathbf{A})^{-1}\mathbf{B}^T$ to the recursive Eq. (8), since during the updating step of the inverse of the correlation matrix only the new sample term must be inverted. Identifying the corresponding terms in the lemma with our problem $\mathbf{R}^{-1}(t) = ((1-\alpha)\mathbf{R}(t-1) + \alpha\mathbf{x}(t)\mathbf{x}^T(t))^{-1}$, the resulting recursive updating rule for the inverse of the sample correlation matrix is:

$$\mathbf{R}^{-1}(t) = \frac{1}{1-\alpha}\left(\mathbf{R}^{-1}(t-1) - \frac{\alpha\mathbf{R}^{-1}(t-1)\mathbf{x}(t)\mathbf{x}^T(t)\mathbf{R}^{-1}(t-1)}{1+\alpha\left(\mathbf{x}^T(t)\mathbf{R}^{-1}(t-1)\mathbf{x}(t)-1\right)}\right) \tag{10}$$

4 Recursive least squares algorithm

In the previous section, we have seen how the theoretical linear regression solution based on the normal equations can be transformed in a practical algorithm for data streams using iterative estimates of the correlations. This approach is closely related to the least squares (LS) problem. In the LS formulation, the sum of squares of the errors is minimized.

$$\varepsilon_{LS}(L) = \sum_{t=1}^{L}\left(y(t) - \mathbf{c}^T(L)\mathbf{x}(t)\right)^2. \tag{11}$$

Note that the solution is the same that the MSE solution when the true correlations are substituted by their sample estimates. But in the LS formulation everything is deterministic; we do not explicitly state a stochastic underlying process.

In data streams, it is usual to consider some drift to represent the dynamics of the undergoing process. Therefore, it is

interesting to formulate the regression problem in a way that can incorporate the non-stationarity of the data. The most common solution is the exponentially weighted recursive least squares RLS formulation [4], where a forgetting factor λ is introduced in Eq. (11) to model the memory of the system.

$$\varepsilon_{RLS}(L) = \sum_{t=1}^{L} \lambda^{L-t} \left| y(t) - \mathbf{c}^T(L)\mathbf{x}(t) \right|^2. \tag{12}$$

The cost function in Eq. (12) is a modification of the LS cost function given in Eq. (11) to incorporate the forgetting factor $0 \ll \lambda < 1$ so recent samples are more important than the old ones in the error computation. The exponential factor allows to obtain a weighted correlation matrix in order to accommodate the time-varying nature of the signals. In other words, $\lambda = 1 - \alpha$ in previous Eqs. (8), (9) and (10). The correctness of the RLS relies on the invertibility of the input correlation matrix $\mathbf{R}(t)$ and on the forgetting factor λ is chosen close to 1 for systems with slowly varying parameters, ensuring stability.

The main advantage of the RLS is its flexibility and simplicity, since no model about λ is required. The same characteristic can become a disadvantage when the dynamics of the process are accurately modeled by more complex systems, such as state-space models with Kalman filters [25]. In fact, RLS can be seen as a particular state-space model where the state variable is constant and the noise is heteroschedastic with exponentially time-varying variance [20].

The solution of the RLS problem involves also the inversion of a matrix, but it can be avoided as we did before. Expressed in terms of the forgetting factor λ instead of α and $\mathbf{P}(t) = \mathbf{R}^{-1}(t)$, one step of the basic RLS algorithm consists of:

$$\begin{aligned}
\mathbf{P}(t) &= \frac{1}{\lambda}\left(\mathbf{P}(t-1) - \frac{\mathbf{P}(t-1)\mathbf{x}(t)\mathbf{x}^T(t)\mathbf{P}(t-1)}{\lambda + \mathbf{x}^T(t), \mathbf{P}(t-1)\mathbf{x}(t)}\right) \\
\mathbf{r}(t) &= \lambda \mathbf{r}(t-1) + y(t)\mathbf{x}(t) \\
\mathbf{c}(t) &= \mathbf{P}(t)\mathbf{r}(t) \\
\hat{y}(t) &= \mathbf{c}^T(t)\mathbf{x}(t).
\end{aligned} \tag{13}$$

4.1 Recursive least squares for data streams

The implementation of the RLS algorithm given by Eq. (13) has some drawbacks from a data stream point of view. First, the algorithm is not outputting an answer as soon as the input vector arrives since the prediction depends on c(t) which depends on y(t). In a stream mining scenario, the predicted output $\hat{y}(t)$ must be provided immediately when the new instance $\mathbf{x}(t)$ is available (test-then-train or prequential evaluation). This can be solved easily by rearranging computations, calculating first $\hat{y}(t)$ using $c(t-1)$ instead of $c(t)$. Second, we introduce some auxiliary vectors to reduce memory and computational requirements. The final version of the RLS algorithm for data streams reads:

$$\begin{aligned}
\hat{y}(t) &= \mathbf{c}^T(t-1)\mathbf{x}(t) \\
e(t) &= y(t) - \mathbf{c}^T(t-1)\mathbf{x}(t) \\
\hat{\mathbf{g}}(t) &= \mathbf{P}(t-1)\mathbf{x}(t) \\
\beta(t) &= \lambda + \hat{\mathbf{g}}^T(t)\mathbf{x}(t) \\
\mathbf{g}(t) &= \hat{\mathbf{g}}(t)/\beta(t) \\
\mathbf{P}(t) &= \lambda^{-1}\left(\mathbf{P}(t-1) - \mathbf{g}(t)\hat{\mathbf{g}}^T(t)\right) \\
\mathbf{c}(t) &= \mathbf{c}(t-1) + \mathbf{P}(t)\mathbf{x}(t)e(t).
\end{aligned} \tag{14}$$

where only the upper part of $\mathbf{P}(t)$ must be calculated since it is a symmetric matrix. The typical initialization values are $\mathbf{c}(0) = 0$ and $\mathbf{P}(0) = \delta^{-1}\mathbf{I}$ with δ a small value if no prior knowledge about the correlation matrix is available. An easy way to interpret the algorithm is to analyze the coefficients updating rule $\mathbf{c}(t) = \mathbf{c}(t-1) + \mathbf{P}(t)\mathbf{x}(t)e(t)$. It means that the filter is moved in the direction of the gain vector $\mathbf{P}(t)\mathbf{x}(t)$ by an amount controlled by the error $e(t)$, while the $\mathbf{P}(t)$ adaptation does not depend on the filtering process but only on the input $\mathbf{x}(t)$ (estimated correlations), and where the λ factor modulates the time-varying response of the whole system.

The algorithm in Eq. (14) follows the test-then-train paradigm since the response is provided immediately (first line in Eq. (14)). This is called the a priori error $e(t)$ in adaptive filtering literature and is equivalent to prequential paradigm in data streams literature. In adaptive filtering nomenclature, the a posteriori error is defined as the error after the filter is updated.

$$\epsilon(t) = y(t) - \mathbf{c}^T(t)\mathbf{x}(t).$$

The second drawback in RLS adaptive filtering is the computational cost; although the matrix inversion is replaced by a scalar division, every step in Eq. (14) requires $O(2N^2 + 4N)$ operations [19].

This is because RLS is related to Gauss–Newton optimization that involves the solution of the problem in one single step using the inversion of the Hessian. Alternatively to the RLS approach that solves the problem in a fast but high computational cost way, we can use traditional gradient-based optimization methods that avoid the Hessian matrix but they do not achieve the solution in one single step.

5 Least mean squares algorithm

The main idea behind any adaptive methods is that the new estimate is obtained from the previous one moving in the direction that the MSE is minimized. The simplest idea is

to consider the stochastic version of Eq. (3) by removing the expectation operator and using the one sample estimate, i.e., the instantaneous error. The most famous algorithm following this idea is the least mean square (LMS), a recursive gradient descent algorithm looking for the coefficients that minimize the instantaneous MSE [29].

Since the negative gradient vector points in that direction, we just have to calculate the partial derivatives of the MSE with respect to the coefficients of the system and update.

$$\mathbf{c}(t+1) = \mathbf{c}(t) - \mu \nabla \varepsilon_{MSE}(t). \tag{15}$$

where $\nabla \varepsilon_{MSE}(t)$ is the gradient and μ the step size.

Calculating the gradient vector, the LMS algorithm is obtained.

$$\begin{aligned} \hat{y}(t) &= \mathbf{c}^T(t-1)\mathbf{x}(t) \\ e(t) &= y(t) - \mathbf{c}^T(t-1)\mathbf{x}(t) \\ \mathbf{c}(t) &= \mathbf{c}(t-1) + \mu \mathbf{x}(t) e(t). \end{aligned} \tag{16}$$

where $e(t)$ is the instantaneous error at time t. For correctness, the step size μ must satisfy the condition $0 < \mu < \frac{2}{\lambda_{\max}}$, where $\lambda_{\max}$ is the largest eigenvalue of the input correlation matrix $\mathbf{R}(t)$. It can track time-varying coefficients as long as the rate of variation is slower than the adaptation rate determined by μ.

The main advantage of the LMS is its low computational complexity compared to the RLS; it also converges in a stationary scenario and is stable even with finite precision arithmetic. The drawback is that convergence can be very slow depending on the input data and initialization.

One way to increase the convergence speed of the LMS algorithm is to use a changing step size $\mu(t)$. This is achieved by normalizing with respect to the instantaneous input signal power $\mu(t) = \mu/(\omega + \mathbf{x}^T(t)\mathbf{x}(t))$, where ω is a small constant term to avoid instability problems when the input signal is very small. This algorithm is called the normalized LMS (NLMS).

$$\begin{aligned} \hat{y}(t) &= \mathbf{c}^T(t-1)\mathbf{x}(t) \\ e(t) &= y(t) - \mathbf{c}^T(t-1)\mathbf{x}(t) \\ \mathbf{c}(t) &= \mathbf{c}(t-1) + \frac{\mu}{\omega + \mathbf{x}^T(t)\mathbf{x}(t)} \mathbf{x}(t) e(t). \end{aligned} \tag{17}$$

The NLMS has the advantage with respect to basic LMS that the error is minimized faster and the disadvantage that it requires an additional vector multiplication.

A different way to modify the basic LMS algorithm is to use the sign function in the updating rule of the filter coefficients. The sign-error LMS algorithm uses the sign of the error instead of the error.

$$\begin{aligned} \hat{y}(t) &= \mathbf{c}^T(t-1)\mathbf{x}(t) \\ e(t) &= y(t) - \mathbf{c}^T(t-1)\mathbf{x}(t) \\ \mathbf{c}(t) &= \mathbf{c}(t-1) + \mu \mathbf{x}(t) \text{sign}(e(t)). \end{aligned} \tag{18}$$

If we apply the sign function to the data, we obtain the sign-data LMS method.

$$\begin{aligned} \hat{y}(t) &= \mathbf{c}^T(t-1)\mathbf{x}(t) \\ e(t) &= y(t) - \mathbf{c}^T(t-1)\mathbf{x}(t) \\ \mathbf{c}(t) &= \mathbf{c}(t-1) + \mu \,\text{sign}(\mathbf{x}(t)) e(t). \end{aligned} \tag{19}$$

And if we apply the sign function to both of them, we obtain the sign-sign LMS algorithm.

$$\begin{aligned} \hat{y}(t) &= \mathbf{c}^T(t-1)\mathbf{x}(t) \\ e(t) &= y(t) - \mathbf{c}^T(t-1)\mathbf{x}(t) \\ \mathbf{c}(t) &= \mathbf{c}(t-1) + \mu \,\text{sign}(\mathbf{x}(t)) \,\text{sign}(e(t)). \end{aligned} \tag{20}$$

Another option to improve the convergence is to use the LMS Gauss–Newton algorithm that exploits the use of the sample covariance matrices to control the direction during the updating step (improvement in convergence speed vs. increasing computational complexity). The algorithm is more complicated, since in every iteration the inverse of the covariance matrix must be also updated. Nevertheless, the inversion of the matrix is avoided thanks to the matrix inversion lemma, reducing the computational cost.

$$\begin{aligned} \hat{y}(t) &= \mathbf{c}^T(t-1)\mathbf{x}(t) \\ e(t) &= y(t) - \mathbf{c}^T(t-1)\mathbf{x}(t) \\ \mathbf{P}(t) &= \frac{1}{1-\alpha}\left(\mathbf{P}(t-1) - \frac{\mathbf{P}(t-1)\mathbf{x}(t)\mathbf{x}^T(t)\mathbf{P}(t-1)}{\frac{1-\alpha}{\alpha} + \mathbf{x}^T(t)\mathbf{P}(t-1)\mathbf{x}(t)}\right) \\ \mathbf{c}(t) &= \mathbf{c}(t-1) + \mu \mathbf{P}(t)\mathbf{x}(t)e(t). \end{aligned} \tag{21}$$

where $\mathbf{P}(t)$ is an estimate of the inverse of the correlation matrix as we showed before. The LMS Gauss–Newton algorithm in Eq. (21) is the same as the RLS in Eq. (14) when $\mu = \alpha = 1 - \lambda$, being λ the forgetting factor in the RLS. The difference with the basic LMS algorithm is due to the $\mathbf{P}(t)$ matrix. So, if this matrix is equal to the identity matrix, both algorithms are the same.

In conclusion, there is a trade-off between computational burden and performance. The ideal would be to reduce the computational complexity of the algorithms presented so far, specially the RLS, but maintaining the convergence speed of both the RLS and the NLMS. In the next section, we introduce an idea that reduces the number of computations.

6 New adaptive filtering algorithm for data streams

In previous sections, we have presented the most common linear adaptive filtering algorithms and discussed their customization for data stream scenarios. Now, in this section, we will delve deeper by proposing a new algorithm. One common goal in data streams is to reduce the computational load and memory resources. The memory requirements can be reduced using auxiliary variables. For example, in the RLS algorithm in Eq. (14), some ancillary vectors such as $\mathbf{g}(t)$ and $\hat{\mathbf{g}}(t)$ are used to reduce the computational cost. They allow to divide the updating rules in simpler steps using previous calculated values. The memory requirements are low since we only need to keep the coefficients vector, i.e., the filter coefficients. This value is equal to the dimension of the feature vector. This is an advantage compared to ensembles of decision trees, such as ARF-Reg [9] and SOKNL [26], which has to maintain all the tree structures in memory.

Regarding the number of computations in the RLS algorithm, depending on the dimensions of the model, i.e., the feature vector dimension N, the updating of the $\hat{\mathbf{g}}(t)$ vector can be problematic since it involves a matrix multiplication. But as we mentioned previously, the purpose of all these updating rules is to obtain a gain vector that is regularized by the error term $e(t)$. The same happens with the LMS-based algorithms: The change of the vector of coefficients is controlled by the error term. It means that when this error is small or when it is of the same order than the system noise, the updating rules are not necessary since to change the filter does not guarantee an increment in the signal-to-noise ratio of the system. This is specially relevant in unknown and complex scenarios as usually it happens with data streams, where the filter must react to fast changes in the input signals. In other words, we can skip all the computations in all previously LMS, NLMS and RLS explained algorithms but the first two lines where the output and error are computed. If the error is lower than certain threshold, the filter and associated variables are not updated, reducing on average the number of computations. This is of paramount importance for the case of RLS, since it is the most computationally demanding algorithm. It converges more quickly than LMS but requires more computations per instance. So it would be great if we could obtain a version of the RLS algorithm that retains its fast convergence characteristic. At the same time, it should reduce the number of times the updating equations are used. By employing the concept of considering only the instances where the error is above a certain value, we obtain such an algorithm.

In this way, the algorithms work as drifting detectors; for every new observation, first the output $\hat{y}(t) = \mathbf{c}^T(t-1)\mathbf{x}(t)$ is provided as mandatory in data streams; second, the error $e(t)$ is computed; and third, this error is compared to some threshold value ζ to decide whether it is worth running the updating steps. If the error is smaller than the upper bound limit threshold ζ, i.e., $\left|y(t) - \mathbf{c}^T(t)\mathbf{x}(t)\right| \leq \zeta$, the algorithm variables are not updated, and the same parameter vector is used for the next instance, $\mathbf{c}(t) = \mathbf{c}(t-1)$. On the contrary, if the error is large enough, $\left|y(t) - \mathbf{c}^T(t)\mathbf{x}(t)\right| > \zeta$, the updating steps are run and a new filter is obtained $\mathbf{c}(t) \neq \mathbf{c}(t-1)$.

There is a trade-off between accuracy and speed when choosing the value of ζ.

Using a large error threshold means that the updating rules are called fewer times, resulting in many instances not being used to update the filter. This will accelerate the response of the algorithm, since basically it only has to calculate the output for each new input instance (a vector multiplication and a threshold comparison).

However, if the filter is not updated for a while, a small drift may go undetected until the accumulated error is sufficiently large. On the other hand, if the error threshold value is small, we will be able to track any drift in a fast way, but the number of computations will not be reduced so much. In the limit case, $\zeta = 0$, we get the original RLS algorithm.

7 Experimental evaluation and discussion

In this section, we show the experimental results obtained by the proposed adaptive filtering solutions and compare to some state-of-the-art regression methods. We analyze not only the performance for different datasets but the time required by each one. We discuss the results and provide some insights about how to define the filter parameters.

The algorithms used in the first experiment are the following ones: the basic LMS (Eq. 16), LMS with an explicit bias term as an example of how increasing the length of the filter from N to $N+1$ (the input vector is accordingly extended by introducing a one value at the top) affects the results, the normalized LMS (Eq. 17), the sign-error LMS (Eq. 18), the sign-data LMS (Eq. 19), the sign-sign LMS (Eq. 20) and the basic RLS (Eq. 14) and the partially updated RLS. Different step sizes for the LMS-based algorithms and forgetting factors for the RLS methods were tested. In addition to the mentioned equations, we provide the pseudocode for each algorithm in appendix.

For comparison purposes, we also report results for some state-of-the-art methods: the Adaptive Random Forest for Regression [9] (AR), Adaptive Model Rules [1] (AMR), Fast and Incremental Model Trees [16] (FIM) and Self-Optimizing K-Nearest Leaves [26] (SOK). Since AR and SOK are ensemble methods, we report experiments using 10 and 30 learners (AR10, SOK10, AR30 and SOK30).

The datasets used in this study are widely recognized benchmarks for data stream regression problems [1, 9, 11]. Table 1 provides an overview of the datasets, including

Table 1 Characteristics of the evaluated datasets

Dataset	#Instances	#Features	Real
Abalone	4977	7	Yes
Bike	17379	12	Yes
House8L	22784	8	Yes
Metro	48204	4	Yes
Pol	15600	48	Yes
Ailerons	13750	40	No
Elevators	16599	18	No
Fried	40768	10	No
Hyper(A)	500000	10	No
Hyper(G)	500000	10	No
Hyper(I)	465935	10	No

the number of instances, features and whether they are real or synthetic. The synthetic datasets include Hyper(A), Hyper(G) and Hyper(I), which simulate different types of concept drift: Abrupt (A) represents instantaneous changes, Gradual (G) involves a transition phase where old and new concepts coexist, and Incremental (I) introduces a sequence of intermediate changes between concepts. Synthetic datasets are designed to evaluate algorithm behavior under controlled drift scenarios and highlight their ability to adapt to evolving data distributions.

For real-world datasets, Abalone contains measurements related to predicting the age of abalone [24]. The Bike dataset captures hourly bike rental counts and weather conditions [6]. House8L represents housing data. Metro reflects traffic patterns in metro systems [15]. The Pol dataset comes from a large telecommunications application [28]. Synthetic datasets other than the Hyper variations, Ailerons, Elevators and Fried originate from well-known benchmarks designed to test regression algorithms under controlled, simulated environments to evaluate model accuracy and robustness without the complexities of real-world noise.

To assess predictive performance in data streams, one must consider not only the metrics used but also how predictions contribute to these metrics. In terms of how predictions are incorporated into metric calculations, we evaluate the performance of each algorithm using two methods: cumulative and windowed prequential evaluation. In cumulative evaluation, each prediction is tested before the model is updated with the true label. This ensures that reported single metrics, such as those presented in tables, reflect the overall performance across the entire stream. This method is particularly well suited for computing aggregate metrics like the coefficient of determination (R^2). For tracking the temporal dynamics of performance, we employ windowed prequential evaluation. This method calculates metrics over a sliding window of recent predictions, offering insights into how performance evolves as the stream progresses [7].

In terms of metrics, there are several ways to measure the performance of regression algorithms. The most common include the root mean square error (RMSE) and the mean absolute error (MAE). In this study, we calculate the RMSE for each dataset and algorithm. However, since the datasets vary widely in the number of instances, features and underlying statistics, we report the coefficient of determination (R^2) instead. This metric, defined as the ratio between the variance explained by the model and the total variance of the data, is a standard measure in regression. The relationship between RMSE and R^2 is straightforward, as they are mathematically related by:

$$R^2 = 1 - \frac{\sum_{t=1}^{T} \left(y(t) - \hat{y}(t)\right)^2}{\sum_{t=1}^{T} \left(y(t) - \bar{y}\right)^2} = 1 - \frac{T \cdot (RMSE)^2}{\sum_{t=1}^{T} \left(y(t) - \bar{y}\right)^2} \tag{22}$$

In this way, we obtain a normalized data-independent measure R^2 and comparisons are easier between datasets. The R^2 value inform us about how much variance in the data is explained by the regressor. Note that a high value does not mean necessarily a good model or a low value a bad regression. For example, a data stream that follows the linear model but has a large residual noise (the innovative part that cannot be predicted) will obtain a small R^2 value, in spite of being the correct model. Also, an overfitted model can obtain a large R^2 value while its model is totally wrong. In our case, it helps to normalize the different scales used to express the RMSE values taking into account the amplitude and dimensions of the corresponding dataset and the degrees of freedom.

A related measure is the adjusted R squared value:

$$R_a^2 = 1 - \frac{(1 - R^2)(T - 1)}{T - p - 1} \tag{23}$$

where p is the dimension of the feature vector. In this way, we normalize the R^2 value taking into account the number of parameters of the model (it is easier to obtain a better fitting model with more degrees of freedom, so the R_a^2 penalizes very complicated models or in our case a high-dimensional feature space). If the number of samples is large enough, both R^2 values are the same. Its value explains how much of the data variance is explained by the predictor, so the larger the value, the better.

A negative adjusted R^2 value means that the model obtained is worse than the mean predictor (the output equals the mean of the output). In the case of a data stream with no drifting, i.e., where the mean value is not changing, it means that the model is completely wrong. For the real datasets, we do not know the variance of the output nor the noise innovative process that cannot be explained by the regressor. It

means that we cannot conclude anything about the goodness of the different methods in absolute terms, i.e., which is a large enough R^2 value to be considered a good predictor. However, we can compare the R^2 values obtained by the different algorithms and at least conclude that the one with the greater value is the one that provides more information.

7.1 Performance analysis

In Table 2, we have the adjusted R^2 results obtained for each dataset and method. We show them graphically in Fig. 1. For better readability, similar algorithms are represented with the same hue (black–gray for LMS, LMSb and NLMS; red–pink for SignE, SignD and SignS). Additionally, to reduce cluttering, we exclude the AR10 and SOK10 solutions and the negative values are truncated in the figure since they represent cases where the corresponding algorithms totally failed.

In general, the proposed algorithms perform well, in particular the RLS algorithm. Analyzing the best method for each dataset (bold numbers in Table 2), the SOK30 achieves the best results in four datasets: Bike, House8L, Pol and Fried.

RLS performs best in three datasets: Ailerons, Hyper(A) and Hyper(G). LMS, LMSb, SignE and AMR are the top performers in the remaining four datasets: Metro, Abalone, Hyper(I) and Elevators, respectively.

By considering two groups of algorithms—those presented in this paper based on linear adaptive filtering and the classical methods used for regression in data streams—we can derive insights from the results. For example, in Fig. 1, we see clearly that for the House8L dataset, all classical regression methods outperform the adaptive linear filtering solutions (all but RLS fail for this dataset). This is because the House8L data is an extremely nonlinear dataset. In this kind of data, where the relationships between the features and the output are impossible to approximate by a linear relation at any time, state-of-the-art methods outperform linear filtering solutions since the linear assumption is too strong. This is a limitation of any linear-based solution.

Another important issue to analyze is if the corresponding algorithm is robust enough to perform well for many different datasets as we are using in this paper. Remember that a negative R_a^2 value means that the algorithm failed.

Figure 1 illustrates that all algorithms except RLS, AR and SOK experienced failures on at least one dataset. This is due to the convergence issues aforementioned for LMS-based algorithms. They are faster than RLS, but they can show some convergence issues as we experienced with some datasets.

Considering only the three methods—RLS, AR and SOK—that never failed, we present in Fig. 2 the average R_a^2 value and standard deviation of these algorithms across all datasets.

Table 2 Adjusted R^2 for each dataset and algorithm

Dataset	RLS	LMS	LMSb	NLMS	SignE	SignD	SignS	AR10	AR30	AMR	FIMT	SOK10	SOK30
Abalone	0.602	0.582	**0.615**	0.597	0.534	0.365	0.307	0.401	0.465	0.564	0.366	0.551	0.511
Bike	0.452	0.592	0.592	0.707	0.449	0.693	0.525	0.722	0.728	0.445	−8.902	0.773	**0.784**
House8L	0.272	−0.863	−0.863	−0.884	−0.899	−0.898	−0.899	0.507	0.527	0.416	0.4	0.548	**0.545**
Metro	0.03	**0.841**	0.517	0.009	0.594	0.747	0.006	0.229	0.254	−2.074	−9.014	0.291	0.202
Pol	0.44	0.352	0.352	0.35	0.228	0.092	−0.131	0.809	0.821	0.658	−0.454	0.854	**0.894**
Ailerons	**0.78**	0.002	0.005	0.316	−0.177	0.675	0.599	0.496	0.53	−24.168	−82.871	0.544	0.415
Elevators	0.263	−1.315	−1.294	−0.912	−1.986	0.138	−0.454	0.314	0.372	**0.508**	−25.303	0.446	0.326
Fried	0.721	0.708	0.707	0.707	0.693	0.412	0.367	0.561	0.632	0.754	0.7	0.779	**0.794**
Hyper(A)	**0.948**	0.948	0.947	0.948	0.94	0.77	0.765	0.431	0.562	0.927	0.937	0.829	0.846
Hyper(G)	**0.919**	0.917	0.917	0.918	0.904	0.751	0.745	0.396	0.523	0.9	0.908	0.79	0.805
Hyper(I)	0.99	0.991	0.991	0.991	**0.993**	0.503	0.502	0.904	0.919	0.899	0.944	0.951	0.946

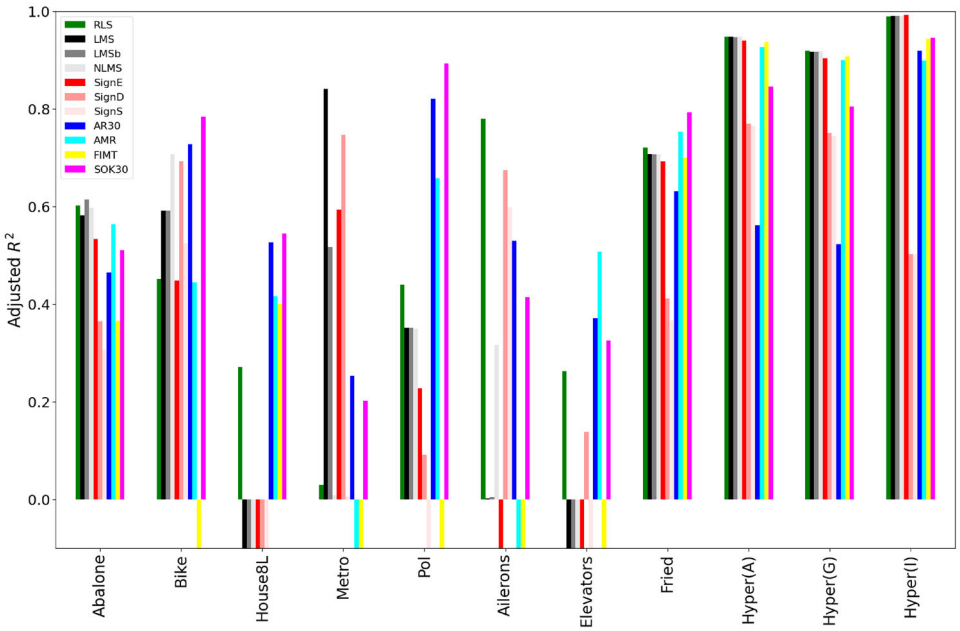


Fig. 1 Adjusted R^2 for each method and dataset

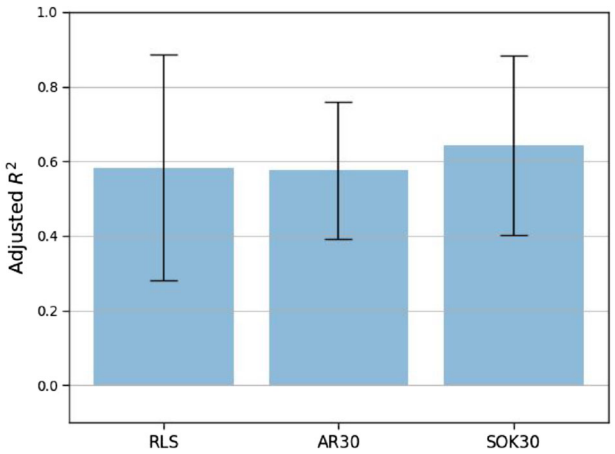


Fig. 2 Average R_a^2 and standard deviation for RLS, AR30 and SOK30 for all datasets

The results indicate similarities among the methods, with a slight advantage observed for the SOK30 method.

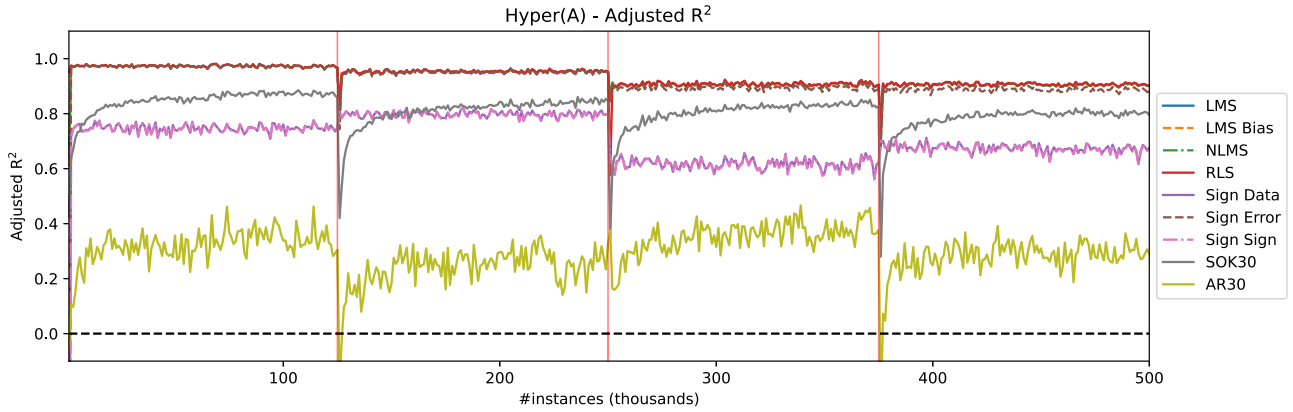
However, it is important to note that the AR and SOK methods are significantly more complex, as evidenced by the comparison of evaluation times for each method.

7.2 Drift performance

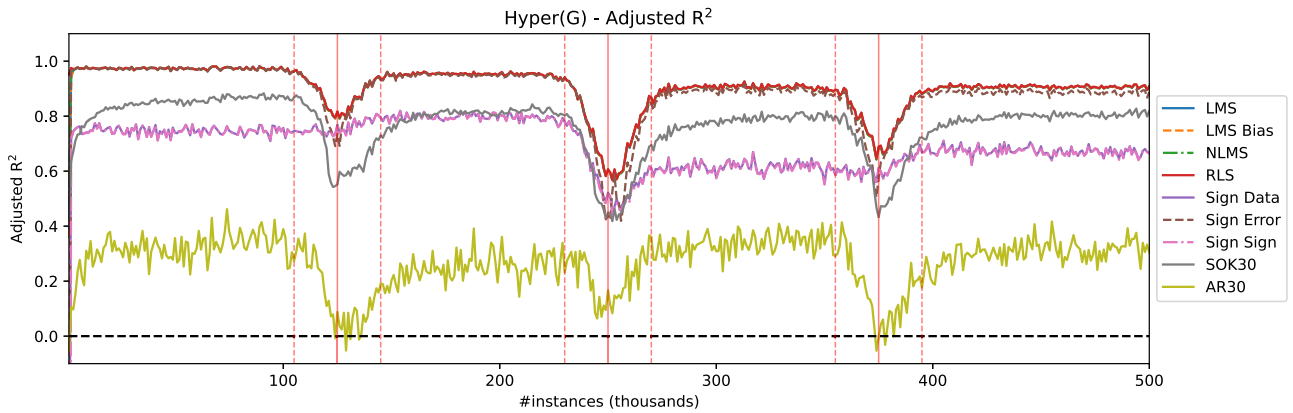
To analyze how the adaptive filter solutions are able to track the corresponding drift in a data stream, in Fig. 3 we show the evolution of the adjusted R^2 by calculating it over tumbling windows of $T/200$ samples for Hyper(A), Hyper(G) and Hyper(I). We select these datasets since the exact location of drifts is known for the Hyper(A) (t=125000, 250000, 375000) and Hyper(G) (same position as in the abrupt case with a transition region of ± 20000 samples). On the other hand, Hyper(I) shows the impact of incremental drift on learning.

Observing Hyper(A) and Hyper(G), we can identify that SOK and AR take longer to recover from drifts than the other algorithms, and this may be explained by the fact that they need to reset several trees that are committed to the previous concept, while the filter adaptation is done implicitly since coefficients are continuously updated. Note in Fig. 3 that some outputs obtained for different algorithms overlap, i.e., they obtain the same regressor. For example, on the one hand, LMS, LMS with bias, NLMS and RLS predict almost the same value for these datasets; on the other hand, the sign-data LMS and the sign-error LMS obtain also similar results (worse than the other adaptive algorithms).

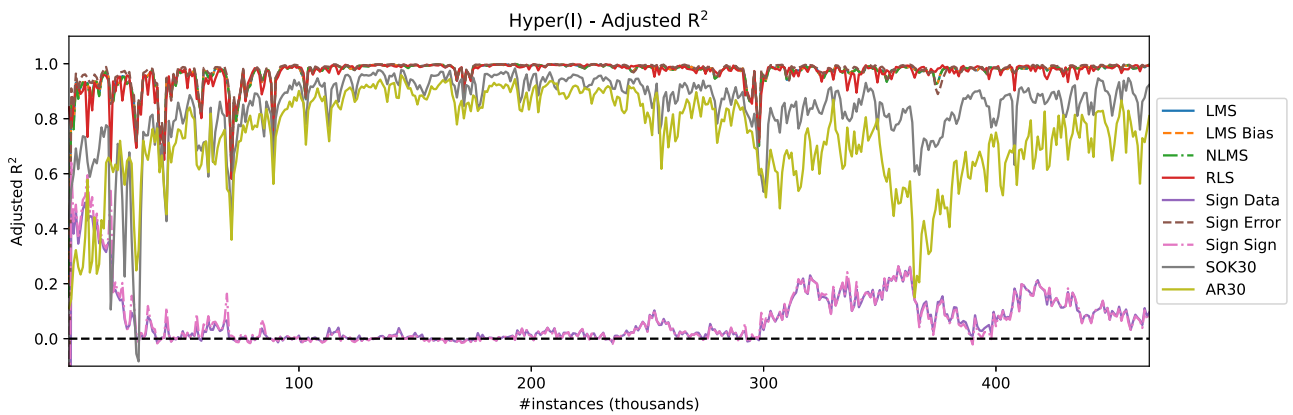
We zoom in for the Hyper(A) data stream for the abrupt drift around instance 125, 000 in Fig. 4 for the RLS, LMS and SOK with 30 learners (SOK30). For the sake of clarity, we do not include the other adaptive filtering solutions since they overlap as we have mentioned before. The size of the window is 50, so we can see an almost real-time response of the different algorithms to the abrupt change; it helps to see how fast the different algorithms are able to detect the drift and adapt the corresponding regressor to the new statistics of the underlying process. We see that the adjusted R^2 is larger for the LMS and RLS algorithms than for the SOK at all times. All algorithms show a similar response to the drift: a transient period where they need to learn the new regressor, in the case of the proposed algorithms, to change the filter coefficients to model correctly the new underlying statistics.



(a) Hyper(A).



(b) Hyper(G).



(c) Hyper(I).

Fig. 3 Adjusted R^2 evolution over the stream. Plotted values are calculated based on a prequential windowed evaluation (window size = 1, 000). Vertical red lines in Hyper(A) and Hyper(G) indicate the known position of a drift. For Hyper(G) the transition phase is indicated by the additional dashed vertical lines

7.3 Forgetting factor influence

In practice, there is a trade-off between the forgetting factor λ used in the RLS algorithm and the abruptness of the drift. If it is too small, the adaptive filter is able to track the changes very fast, but the variance is increased since the underlying correlation matrix estimate is worse. On the contrary, if it is too large, it can take a long time to adapt the filter to the new data distribution when it changes.

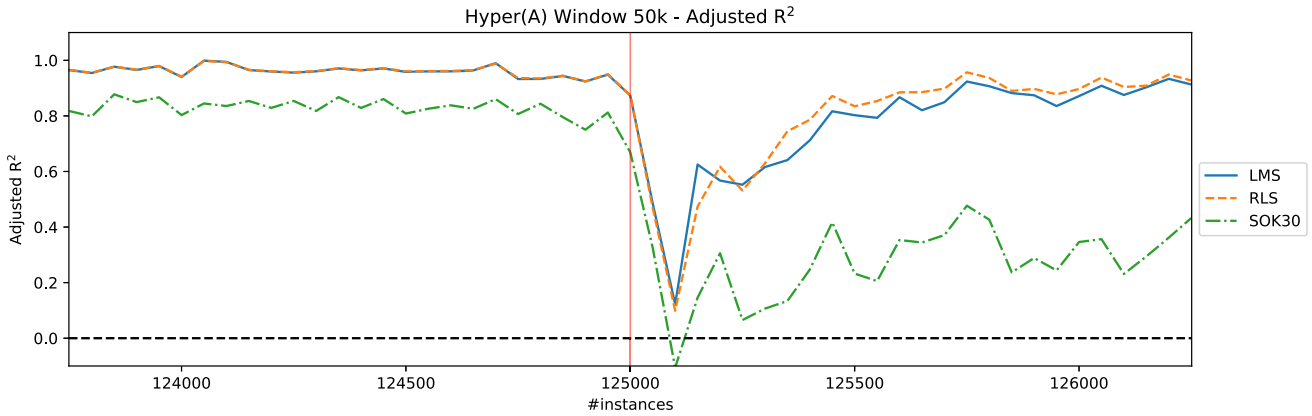


Fig. 4 Adjusted R^2 for a sliding window (size = 50 instances) for the abrupt drift in Hyper(A) at instance 125000 for the RLS ($\lambda = 0.998$, $\delta = 0.2$), LMS and SOK30 solutions. All algorithms are capable of adapting to the concept drifts and converge after a few instances. LMS and RLS error is lower before and after the drift

In Fig. 5, we show the performance for different λ values for the Hyper(A) case around first drift. As we can see, when λ is increased, it takes longer to adapt to the new statistics. For the purpose of clarity, we include the $\lambda = 1$ case to show that this is not an option since it assumes that no drifting is happening, i.e., all previous samples are used (too long of a memory, losing its tracking performance). On the contrary, a very small value means that the solution will have a large variance, since only the last samples are used to estimate the correlation matrix (short of a memory).

7.4 Evaluation time analysis

We have compared and discussed so far the performance of each method. But in real-time applications, it is also important to consider the time computation. In Table 3, we show the evaluation time in CPU seconds for each dataset and algorithm.

It is clear that the adaptive filter solutions are much faster than all the state-of-the-art solutions. Combined with the performance analysis carried out previously, it makes clear that the adaptive filtering paradigm is an adequate approach for data streams. For example, in Fig. 2 we have seen that the average performance for all datasets of the RLS algorithm is similar to ensemble learners AR30 and SOK30. In Table 3, we see that RLS is one or two orders of magnitude faster than any of them. This is a very remarkably advantage of RLS with respect to ensemble-based solutions. The filtering approach benefits from the simplicity of the model and the matrix notation allowing a fast computation in addition to a fast adaptation to changes in the data statistics.

Compared to AMR and FIM methods, RLS is a little bit faster, but in the same order of magnitude. Comparing the time computation for the different filter solutions, as we expected, the RLS algorithm is the slowest one. LMS and sign versions of it are much more faster than the rest of methods. But as we have seen they are also more prone to convergence issues. The ideal will be to keep the RLS performance and the LMS speed. We proposed in Sect. 6 a technique to reduce the number of computations for the RLS algorithm pursuing this goal.

7.5 Fast RLS results

The last experiment analyzes what happens when the updating rules for the RLS algorithm are only applied if the error is greater than a predefined noise level. In Table 4, we show the performance obtained by the partially updated RLS algorithm for different threshold error values. We show the resulting R_a^2 for different percentages of times that the updating rule is applied and the filter coefficients updated. For example, the results for the 10% instances column corresponds to the case where the number of computations is reduced to the square root of the size of the dataset, so the average number of computations become $O(N)$ instead of $O(N^2)$ for the partially updated RLS algorithm.

Table 4 presents the evaluation time for each case, expressed as a percentage of the time required when the filter is updated for every feature input. The time for updating the filter for 100% of instances is provided in Table 3.

For example, in the Hyper(I) dataset (last row), updating the filter for only 10% of the instances reduces the evaluation time by half (50.37%), requiring only 1.60 s instead of 3.19 s (Hyper(I)-RLS value in Table 3).

As we expected, for some of the datasets, when reducing the number of instances that are used to update the filter values, the R_a^2 decreases; e.g., for Abalone dataset, the value is reduced from $R_a^2 = 0.6018$ to $R_a^2 = 0.4976$ when using only 10% of the instances. On the other hand, it takes only 36% of the time compared to the case where all instances

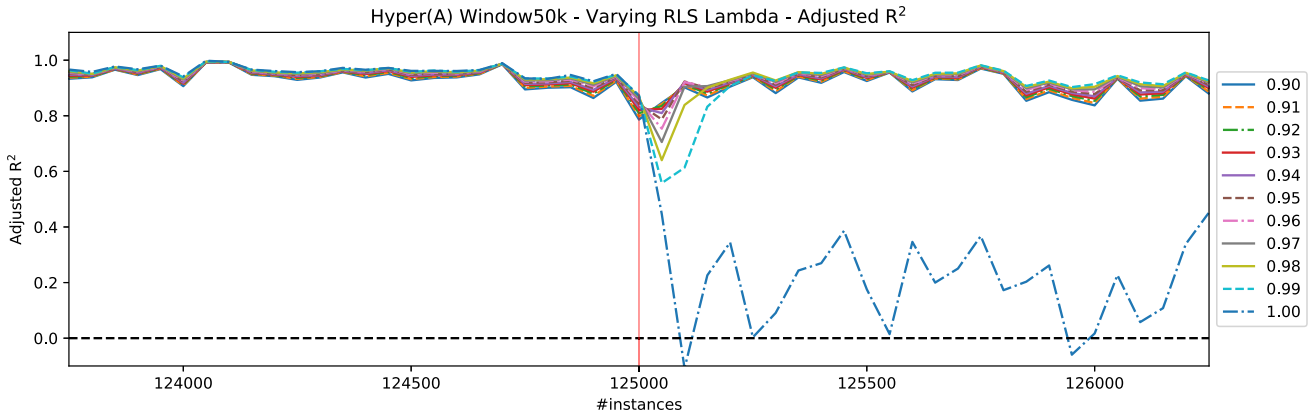


Fig. 5 Adjusted R^2 for a sliding window (size = 50 instances) for the abrupt drift in Hyper(A) at instance 125000 for the RLS varying λ from 0.90 to 1.0 ($\delta = 0.2$)

Table 3 Evaluation time (CPU seconds) for each dataset and algorithm

Dataset	LMS	LMSb	NLMS	SignE	SignD	SignS	RLS	AR10	AR30	AMR	FIM	SOK10	SOK30
Abalone	0.07	0.07	0.06	0.08	0.08	0.06	0.17	0.86	1.29	0.25	0.04	0.7	1.22
Bike	0.06	0.06	0.06	0.06	0.06	0.06	0.11	1.17	2.95	0.62	0.13	1.19	4.45
House8L	0.04	0.04	0.04	0.04	0.04	0.04	0.11	1.55	5.51	0.56	0.16	2.2	7.7
Metro	0.05	0.06	0.06	0.06	0.06	0.06	0.2	1.79	5.61	0.97	0.14	2.14	7.22
Pol	0.09	0.1	0.09	0.1	0.1	0.1	0.55	1.41	5.75	0.63	0.28	2.17	6.4
Ailerons	0.1	0.08	0.09	0.09	0.09	0.08	0.35	2.38	7.03	0.59	0.24	2.42	7.82
Elevators	0.05	0.06	0.05	0.05	0.05	0.05	0.18	0.94	2.72	0.49	0.12	0.92	3.62
Fried	0.07	0.08	0.09	0.08	0.08	0.08	0.24	2.21	7.66	1.08	0.38	2.79	8.93
Hyper(A)	1.2	1.16	1.19	1.16	1.22	1.18	3.25	41.25	149.5	14.51	7.79	82.1	262.54
Hyper(G)	1.14	1.15	1.15	1.23	1.17	1.19	3.37	46.23	149.39	13.66	7.92	81.89	263.08
Hyper(I)	1.21	1.06	1.11	1.11	1.12	1.1	3.19	43.16	147.06	13.92	5.73	47.9	168.16

Table 4 Adjusted R^2 for each dataset for different percentage of iterations where the RLS filter is updated and evaluation time in % with respect to the 100% case

Dataset	10% instances		25% instances		50% instances		100%
	R_a^2	Time %	R_a^2	Time %	R_a^2	Time %	R_a^2
Abalone	0.4976	36.80	0.5721	50.79	0.5959	57.29	0.6018
Bike	0.2662	45.20	0.4002	53.33	0.4484	69.22	0.4516
House8L	−0.0762	44.06	0.2205	49.89	0.2738	67.86	0.2722
Metro	0.0205	51.24	0.0225	65.77	0.0261	77.15	0.0302
Pol	−0.3173	29.28	−0.1807	42.57	−0.7907	64.92	0.4403
Ailerons	0.7746	29.53	0.7910	39.33	0.7934	58.60	0.7796
Elevators	0.0911	40.74	0.3287	49.06	0.4000	67.03	0.2535
Fried	0.7079	44.73	0.7184	53.40	0.7209	70.32	0.7207
Hyper(A)	0.9034	55.95	0.9228	66.26	0.9462	90.92	0.9487
Hyper(G)	0.8737	56.83	0.8883	66.55	0.9158	88.54	0.9192
Hyper(I)	0.6908	50.37	0.6992	63.51	0.7021	78.46	0.7032

are used. In conclusion, a trade-off between speeding up and performance must be done for those datasets. It is remarkable that for some other datasets, Ailerons, Fried and Hyper experiments, the performance is not reduced, while reducing enormously the computational time. For example, for the Ailerons dataset, we can achieve a very good performance $R_a^2 = 0.7746$ in only 29% of the time required to obtain almost the same performance $R_a^2 = 0.7796$. This is an effect of the random nature of a data stream, including possible outliers and complicated data generators that as a result move the

filter in a direction that is not correct for the next instance. Of course, this effect is reduced when datasets follow a regular, high signal-to-noise ratio and stable generation process. But as we show for these datasets, many of them benefit from this new error-based approach: If the error is small, this instance does not provide enough evidence to modify the filter coefficients. This signal-to-noise ratio idea can be exported to any data stream algorithm, since it prevents to overlearning when the data noise is similar to the noise of the classification or regression system.

8 Future work and limitations

In this paper, we have focused on the instantaneous linear regression problem stated by Eq. (2). But, for some datasets, such as the bike, the output depends not only on the current input but also on previous inputs and outputs. The model in Eq. (2) can be easily extended to include these input–output time dependencies:

$$y(t) = \sum_{k=1}^{p} a_k(t) y(t-k) + \sum_{k=0}^{q} \mathbf{c}_k^T(t)\mathbf{x}(t-k) \tag{24}$$

where scalars $a_1(t), \ldots, a_p(t)$ and vectors $\mathbf{c}_1(t), \ldots, \mathbf{c}_q(t)$ must be estimated. If the autoregresive part does not exist, i.e., $y(t)$ does not depend on previous output values and Eq. (24) is reduced to $y(t) = \sum_{k=0}^{q} \mathbf{c}_k^T(t)\mathbf{x}(t-k)$, the algorithms are the same as those presented in this paper. The only difference is that the feature vector size is increased in order to include previous input values, with memory and computational resources increasing accordingly.

If the current output depends on previous output values, the algorithms must be modified as the new recursion term complicates the problem. The major drawback in that case is that stability issues can arise and some assumptions must be usually made in order to obtain a practical algorithm, e.g., that the $a_k(t)$ coefficients do not vary too fast.

Another important consideration in the case of data streams is that storing previous output values increases the required memory. We are currently working on this issue in order to find optimal implementations for data streams.

The primary limitation arises from the linear assumption. The linear model in Eq. (24) can be modified to include a nonlinear function, obtaining the so called Hammerstein structure [23, 30], that relates the input signals $\mathbf{x}(t)$ with the output of the system $y(t)$:

$$\mathbf{y}(t) = \sum_{k=1}^{p} a_k(t) y(t-k) + \sum_{k=0}^{q} \mathbf{c}_k(t) f(\mathbf{x}(t-k)) \tag{25}$$

where $f(.)$ is the nonlinear function applied to the input signals.

The model in Eq. (25) can be seen as two systems connected in series. The first one (the Hammerstein structure) linearizes the input data using nonlinear scalar functions, e.g., the square root or the log function; the second one is a dynamic linear system that uses as input the output of the nonlinear static system. Depending on the nature of the data, in some cases it is easier to apply the inverse nonlinear function to the result variable $y(t)$ and work with a linear model in the transformed domain, for example, $\log y(t)$.

Thus, transforming the nonlinear problem into a linear one allows the application of all the algorithms discussed in this paper, ensuring a rapid response. Obviously, if the relationship between inputs and output is much more complex than a general nonlinear function such as log, results will get worse (trade-off between accuracy and complexity of the model (time computation)). In summary, the adaptive linear filter-based modeling of the problem can be extended to include more complex structures such as in Eq. (25). Alternatively, a model-free solution based exclusively on available data would not require any assumption at all about the data distributions. However, such solutions as the ones based on regression trees or neural networks would require much more data and computational time as we have shown in previous experiments.

9 Conclusions

This paper introduces adaptive linear filtering techniques as a novel addition to current regression methods for data streams. The major advantage is that they require less resources and can adapt in a fast way to any drift in the data since adaptation is implicit in the updating process. The major drawback is the linear assumption that can be too strong for some datasets.

Surprisingly, these techniques have yet to gain popularity in streaming regression.

We introduced algorithms with various optimization procedures and discussed how they can be adapted and accelerated for the data stream scenario. We presented an extensive empirical evaluation comparing adaptive filters with the state-of-the-art data stream regression methods. Adaptive Filtering seems a promising way to deal with data stream learning. In the future, we will explore the introduction of time dependencies and efficient implementations in order to keep a good performance at low computational cost.

A Pseudocodes of the LMS and RLS algorithms

In all algorithms, the input is the data stream and the previous target. Depending on the algorithm, some parameters are required and must be initialized.

For the family of LMS-based algorithms, the pseudocode is shown in Algorithm 1.

Algorithm 1 Family of LMS-based algorithms

Inputs: data stream x, previous target y'
Outputs: filter c, prediction y
Initialize: $\mathbf{c} = 0, \mu, \omega$
while x do
  $y = \mathbf{c}^T\mathbf{x}$
  $e = y' - y$
  switch *algorithm*
   case LMS: $\mathbf{c} \leftarrow \mathbf{c} + \mu\mathbf{x}e$
   case NLMS: $\mathbf{c} \leftarrow \mathbf{c} + \mu\frac{\mathbf{x}}{\omega+\mathbf{x}^T\mathbf{x}}e$
   case sign-error: $\mathbf{c} \leftarrow \mathbf{c} + \mu\mathbf{x}\text{sign}(e)$
   case sign-data: $\mathbf{c} \leftarrow \mathbf{c} + \mu\,\text{sign}(\mathbf{x})e$
   case sign-sign: $\mathbf{c} \leftarrow \mathbf{c} + \mu\,\text{sign}(\mathbf{x})\,\text{sign}(e)$
  end switch
end while
end

where the learning rate $0 < \mu < 1$ must be tuned depending on the problem. A large value allows a fast adaptation to any drift in the input data but results in a larger variability in the output. For the NLMS algorithm, the additional parameter ω can be any small value to make sure that the division is well defined for very small input instances, e.g., $\omega = 10^{-13}$.

For the RLS algorithm, the pseudocode is shown in Algorithm 2.

Algorithm 2 Family of RLS-based algorithms

Inputs: data stream x, previous target y'
Outputs: filter c, prediction y
Initialize: $\mathbf{c} = 0, \mathbf{P} = 10^{-2}\mathbf{I}, \lambda, \zeta$
while x do
  $y = \mathbf{c}^T\mathbf{x}$
  $e = y' - y$
  if $e > \zeta$ **then**
    $\hat{\mathbf{g}} = \mathbf{P}\mathbf{x}$
    $\beta = \lambda + \hat{\mathbf{g}}^T\mathbf{x}$
    $\mathbf{g} = \hat{\mathbf{g}}/\beta$
    $\mathbf{P} = \lambda^{-1}\left(\mathbf{P} - \mathbf{g}\hat{\mathbf{g}}^T\right)$
    $\mathbf{c} \leftarrow \mathbf{c} + \mathbf{P}\mathbf{x}e$
  end if
end while
end

Here the if condition is removed for the complete RLS algorithm (when the filter is updated for each instance). The forgetting factor λ establishes the memory of the system; e.g., $\lambda = 1$ means that all instances are equally important while for $\lambda = .95$ the 45th previous instance error only weights 0.1 compared to the current one. The error threshold ζ establishes the number of computations.

Author contributions J.I. (method definition, theoretical arguments, initial code implementation, experiments definitions), H.M.G. (experiments definitions, code review and reimplementations, experiments discussion, text review), B.P. (methodology review, experiments discussion and comments), A.B. (methodology review, text review).

Funding Open Access funding enabled and organized by CAUL and its Member Institutions

Data availability Data is provided within the manuscript or supplementary information files (such as synthetic data).

Declarations

Conflict of interest The authors declare no conflict of interest.

Ethical approval Not applicable.

Humans and animals rights Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Almeida, E., Ferreira, C., Gama, J.: Adaptive model rules from data streams. In: ECML, pp. 480–492. Springer (2013)
2. Bifet, A., Gavalda, R.: Learning from time-changing data with adaptive windowing. In: SIAM international conference on data mining, pp. 443–448 (2007)
3. Bifet, A., Gavaldà, R.: Adaptive learning from evolving data streams. In: International Symposium on Intelligent Data Analysis, pp. 249–260. Springer (2009)
4. Campi, M.C.: Exponentially weighted least squares identification of time-varying systems with white disturbances. IEEE Transact. Signal Proc. **42**(11), 2906–2914 (1994)
5. Domingos, P., Hulten, G.: Mining high-speed data streams. In: ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 71–80 (2000)
6. Fanaee-T, H., Gama, J.: Event labeling combining ensemble detectors and background knowledge. Pro. Artif. Intell. **2**, 113–127 (2014)
7. Gama, J., Sebastiao, R., Rodrigues, P.P.: On evaluating stream learning algorithms. Mach. learn. **90**, 317–346 (2013)
8. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM comput. Surv. (CSUR) **46**(4), 1–37 (2014)

9. Gomes, H.M., Barddal, J.P., Ferreira, L.E.B., Bifet, A.: Adaptive random forests for data stream regression. In: ESANN (2018)
10. Gomes, H.M., Bifet, A., Read, J., Barddal, J.P., Enembreck, F., Pfahringer, B., Holmes, G., Abdessalem, T.: Adaptive random forests for evolving data stream classification. Machine Learning (2017)
11. Gomes, H.M., Montiel, J., Martiello Mastelini, S., Pfahringer, B., Bifet, A.: On ensemble techniques for data stream regression. In: 2020 International Joint Conference on Neural Networks (IJCNN). IEEE (2020)
12. Gomes, H.M., Read, J., Bifet, A., Barddal, J.P., Gama, J.: Machine learning for streaming data: state of the art, challenges, and opportunities. ACM SIGKDD Explorat. Newsletter **21**(2), 6–22 (2019)
13. Haykin, S.S.: Adaptive filter theory. Pearson (2005)
14. Hoeffding, W.: Probability inequalities for sums of bounded random variables. J. Am. Stat. Assoc. (1963). https://doi.org/10.1007/978-1-4612-0865-5_26
15. Hogue, J.: Metro interstate traffic volume (2019). https://doi.org/10.24432/C5X60B. https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume
16. Ikonomovska, E., Gama, J., Džeroski, S.: Learning model trees from evolving data streams. Data Min. know. Discov. **23**(1), 128–168 (2011)
17. Ikonomovska, E., Gama, J., Džeroski, S.: Online tree-based ensembles and option trees for regression on evolving data streams. Neurocomputing (2015). https://doi.org/10.1007/978-1-4612-0865-5_26
18. Ikonomovska, E., Gama, J., Zenko, B., Dzeroski, S.: Speeding-up hoeffding-based regression trees with options. In: ICML, pp. 537–544. Citeseer (2011)
19. Ingle, V.K., Kogon, S.M., Manolakis, D.G.: Statistical and adaptive signal processing. Artech House, Cambridge, MA, USA (2005)
20. Li, T.H.: On exponentially weighted recursive least squares for estimating time-varying parameters and its application to computer workload forecasting. J. Stat. Theory Pract. **2**(3), 339–354 (2008). https://doi.org/10.1080/15598608.2008.10411879
21. Losing, V., Hammer, B., Wersing, H.: Knn classifier with self adjusting memory for heterogeneous concept drift. In: 2016 IEEE 16th international conference on data mining (ICDM), pp. 291–300. IEEE (2016)
22. Mouss, H., Mouss, D., Mouss, N., Sefouhi, L.: Test of page-hinckley, an approach for fault detection in an agro-alimentary production system. In: Asian Control Conference, vol. 2 (2004)
23. Narendra, K., Gallman, P.: An iterative method for the identification of nonlinear systems using a Hammerstein model. IEEE Transact. Automat. Control **11**(3), 546–550 (1966). https://doi.org/10.1109/TAC.1966.1098387
24. Nash, W., Sellers, T., Talbot, S., Cawthorn, A., Ford, W.: Abalone. UCI Machine Learning Repository (1995). https://doi.org/10.24432/C55C7W
25. Sayed, A.H., Kailath, T.: A state-space approach to adaptive rls filtering. IEEE signal processing magazine **11**(ARTICLE), 18–60 (1994)
26. Sun, Y., Pfahringer, B., Gomes, H.M., Bifet, A.: Soknl: a novel way of integrating k-nearest neighbours with adaptive random forest regression for data streams. Data Min. Know. Discov. **36**(5), 2006–2032 (2022)
27. Tylavsky, D.J., Sohie, G.R.: Generalization of the matrix inversion lemma. Proc. IEEE **74**(7), 1050–1052 (1986)
28. Weiss, S.M., Indurkhya, N.: Rule-based machine learning methods for functional prediction. J. Artif. Intell. Res. **3**, 383–403 (1995)
29. Widrow, B., Stearns, S.D.: Adaptive signal processing. Englewood Cliffs, NJ, Prentice-Hall, Inc., 1985, 491 p. **1** (1985)
30. Wills, A., Schön, T.B., Ljung, L., Ninness, B.: Identification of Hammerstein-wiener models. Automatica **49**(1), 70–81 (2013). https://doi.org/10.1016/j.automatica.2012.09.018

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.