UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

INFO-H-515
Big Data Scalable Analytics

Report - Phase 2

Antoine Passemiers

Academic year 2018 - 2019

CONTENTS

1. Introduction 1

2. Methodology 2

2.1 Data generation 2

2.2 Recursive Least Squares (RLS) with forgetting factor 3

2.2.1 First approach – Fully-vectorized version 3

2.2.2 Second approach – Distributed version 3

2.3 Architecture and Spark implementation 4

3. Results 6

3.1 Performance 6

3.2 scalability 10

4. Conclusion 11

1. INTRODUCTION

The objective of this assignment is to implement a scalable online algorithm that is able to perform forecasting in a distributed fashion. More specifically, such algorithm should be able to predict the values of some explained variables based on the values of some explanatory variables by having recourse to linear models. In the framework of this project, the Recursive Least Squares (RLS) algorithm has been privileged as it is capable of learning the coefficients of a linear model incrementally.

The implementation extends the notebooks that have been provided along with the project statement. There are mainly three extensions of these notebooks:

- The coefficients of the underlying linear models are generated randomly using a multivariate Gaussian distribution. The parameters of the later are drawn at random using a normal-Wishart distribution and fixed before the learning process starts. Also, multiple output/explained variables are returned instead of a single one.
- The system is made scalable with respect to the number of models to be run in parallel. Running multiple models in parallel with different hyper-parameters allows for fast validation and computation of the optimal hyper-parameter values. In this project, the only hyper-parameter to be tuned is the forgetting factor of the RLS algorithm.
- The system is made scalable with respect to the number of explained variables. When the number of output variables is low, the RLS update function relies on NumPy vectorization, but is actually run in parallel when the number of output variables exceeds a given threshold. This design choice allows for maximum flexibility.

Data samples are sent by a Kafka [3] producer, retrieved by a Kafka consumer and fed as input to a Spark [4] DStream. The later is then used to update the models' coefficients in a distributed manner.

2. METHODOLOGY

2.1 Data generation

Observations $x \in \mathbb{R}^n$ are drawn from a uniform distribution, while each latent coefficient vector $\beta_j \in \mathbb{R}^n$ is drawn from a multivariate Gaussian distribution. Each vector β_j is a column of the coefficient matrix $\beta \in \mathbb{R}^{n \times m}$ and fixed beforehand. Instead of choosing arbitrary mean vector and covariance matrix as parameters of the multivariate Gaussian distribution, these parameters are drawn at random. Also, the covariance matrix should be positive semi-definite. Therefore, a normal-Wishart is used to generate a random mean vector and a precision matrix. The final covariance matrix is obtained by inverting the sampled precision matrix.

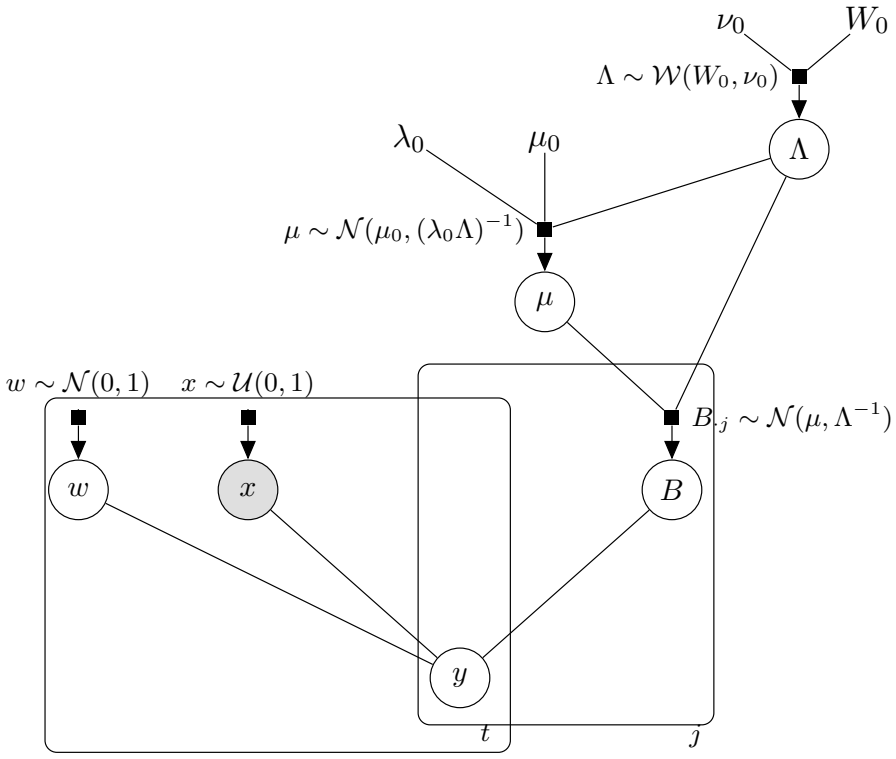

Fig. 2.1: Bayesian network representing the randomly generated samples. Plate notation indicates variable repetition across time and output variables, respectively.

As can be observed from the plate notation in figure 2.1, coefficient vectors $B_{\cdot j}$ are computed only once for each j, and w, x, y are sampled at each time step t, where $y_j = B_{\cdot j} x$. Mean vector μ and precision matrix Λ are sampled only once from a normal-Wishart distribution. Default parameter matrix W_0 is the diagonal matrix, prior mean

vector μ_0 is the zero vector, chosen scaling parameter is 1, and ν_0 has been arbitrarily set to 15.

2.2 Recursive Least Squares (RLS) with forgetting factor

In the standard RLS implementation with forgetting factor, the weights B are estimated incrementally using the following formulas:

$$\begin{cases} V^{(t)} & = \frac{1}{\nu}\left(V^{(t-1)} - \frac{V^{(t-1)}x_t^T x_t V^{(t-1)}}{1+x_t V^{(t-1)}x_t^T}\right) \\ \alpha^{(t)} & = V^{(t)}x_t^T \\ e^{(t)} & = y^{(t)} - x_t\hat{\beta}^{(t-1)} \\ \hat{\beta}^{(t)} & = \hat{\beta}^{(t-1)} + \alpha^{(t)}e^{(t)} \end{cases} \tag{2.1}$$

where $V^{(t)}$ is a matrix of shape $n \times n$, n is the number of explanatory variables and $e^{(t)}$ is the prediction error on the example retrieved from the Kafka consumer at time step t. In this project, the learning algorithm should be able to handle multiple output variables. Therefore, such formulation has to be extended to a multi-output case.

2.2.1 First approach – Fully-vectorized version

This approach is used when the number of output/explained variables is low, and relies on NumPy's efficient implementation of the dot-product. Let $B \in \mathbb{R}^{n\times m}$ denote a matrix this time. Each machine on the cluster computes **the whole coefficient matrix** B with a different forgetting factor.

$$\begin{cases} \alpha_t & = V^{(t)}x_t^T \\ e^{(t)} & = y^{(t)} - x_t\hat{B}^{(t-1)} \\ \hat{B}^{(t)} & = \hat{B}^{(t-1)} + \alpha_t^T e^{(t)} \end{cases} \tag{2.2}$$

$e^{(t)} \in \mathbb{R}^m$ is vector and $e_j^{(t)}$ is the prediction error on the output variable y_j retrieved at time t from the Kafka consumer.

2.2.2 Second approach – Distributed version

The scalability of the algorithm w.r.t. to the number of output variables is enabled by this second approach. However, it is actually interesting when the number of output variables is larger than the number of input variables. At this condition only, the bottleneck of the algorithm is the computation of $\hat{B}^{(t)}$ and not $V^{(t)}$. Therefore, such approach is used only when the number of output variables is sufficiently high. For demonstration purposes, the second approach is used in the notebook since the number

of output variables has been set to 8 and the threshold to 6. Note that this threshold is arbitrary and is a user preference. In practice, I believe that a reasonable design choice would be to set the threshold to the vector size from which NumPy starts having recourse to multithreading.

$$\begin{cases} \alpha_t & = V^{(t)} x_t^T \\ e^{(t)} & = y_j^{(t)} - x_t \hat{B}_{\cdot j}^{(t-1)} \\ \hat{B}_{\cdot j}^{(t)} & = \hat{B}_{\cdot j}^{(t-1)} + \alpha_t^T e^{(t)} \end{cases} \tag{2.3}$$

In the present approach, each machine of the cluster computes one column of the coefficient matrix B. Therefore, each node is assigned an output variable j and re-estimates $\hat{B}_{\cdot j}^{(t)}$ at each time step t. Also, $e^{(t)}$ is a scalar and represents the prediction error on y_j at time step t.

2.3 Architecture and Spark implementation

In figure 2.2, both fully-vectorized and distributed approaches are shown. Note that in both approaches, the algorithm is scalable with respect to the number of models. This means that all models are run in parallel, regardless of the approach used. Transformations in the vectorized approach are represented by plain lines and transformations in the distributed approach are represented by dashed lines. In both approaches, the DStream is first transformed using a map transformation, as indicated by transformation (1). Transformation (1) converts messages received from Kafka to key/value pairs where the key is the name of the model and the value is a NumPy array containing the numerical values present in the messages.

Initial RLS states are all initialized with zero coefficients for matrix B and the identity matrix for matrix V. A full state refers to a state related the whole coefficient matrix B of a model, while a partial state is related to only a specific column of B. In the distributed approach, each initial state is split into m partial states with transformation (3) beforehand. Note that all partial states share the same key as the original full state. These partial states are then updated by performing a RLS step in parallel, as indicated by transformation (4). In the vectorized approach, the full states are updated by performing a RLS step in a vectorized fashion as indicated by transformation (2).

In the distributed approach, full states are reconstructed with a groupByKey operation directly followed by a map operation. These two transformations are represented by the single transformation (5). Transformation (6) is a generic map transformation used for retrieving information and eventually displaying it, like the prediction error. For more details on the implementation of each transformation, please refer to the *KafkaReceiveRLS.ipynb* notebook.

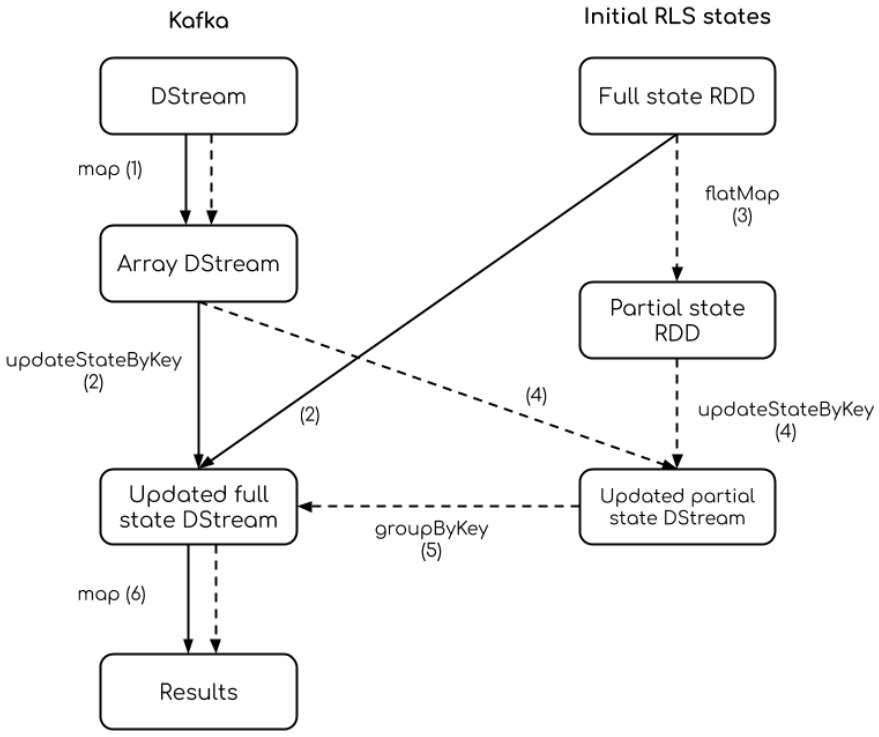

Fig. 2.2: Lineage graph for the proposed architecture. Dashed lines indicate the Spark transformations applied in the distributed version and plain lines indicate the transformations applied in the fully-vectorized version.

The architecture is scalable both with respect to the number of models and output variables:

- Each model is run in parallel since each state is characterized by its key, where each key is associated to a unique model. It must be noted that running multiple models in parallel is an embarassingly parallel task.
- Each output variable is learnt in parallel (in the fully-vectorized approach only) due to the flatMap transformation: each column of the coefficient matrix is assigned a differnent partial state and all columnsassociated to the same model are assigned the same key. All partial states sharing the same key are then grouped using a groupByKey transformation.

3. RESULTS

3.1 Performance

Performance of the RLS algorithm in terms of prediction accuracy has been assessed offline for reasons of expediency, using the notebook *AssessPerformanceOffline.ipynb*. Figure 3.1 shows the error on example (x_t, y_t) as a function of the simulation step t. For readability purposes, the curves have been smoothed using a moving average with a window size of 24.

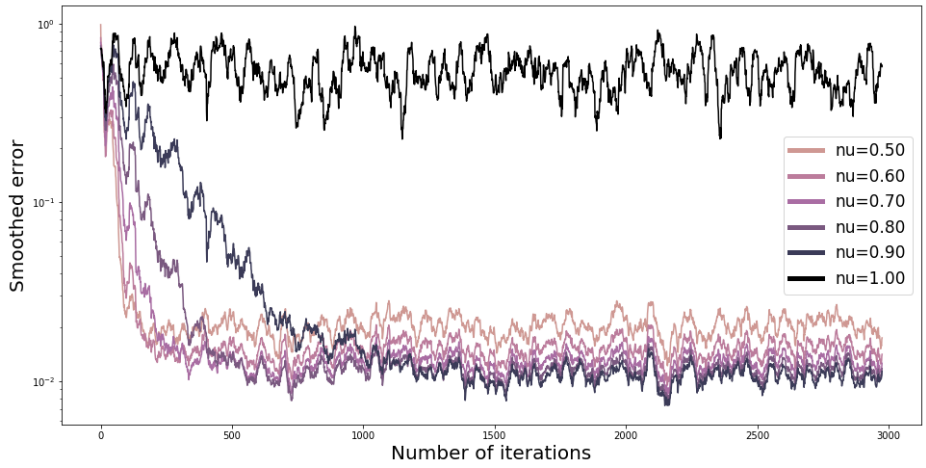

Fig. 3.1: Convergence of RLS models with different forgetting factors, in log-scale. Darker curves are associated to models with higher forgetting factors.

As can be observed, models with higher forgetting factors exhibit slower convergence but yield lower average errors in the long run. Also, a forgetting factor of 1 seems to completely degrade the model's learning capabilities.

Since the simulations are made offline, the ground-truth coefficients are available and can be used to better assess convergence. In figure 3.2, the error has been replaced by the Frobenius norm of the difference between learnt and ground-truth coefficient matrices. Differences in performance between the different forgetting factors can be observed more clearly.

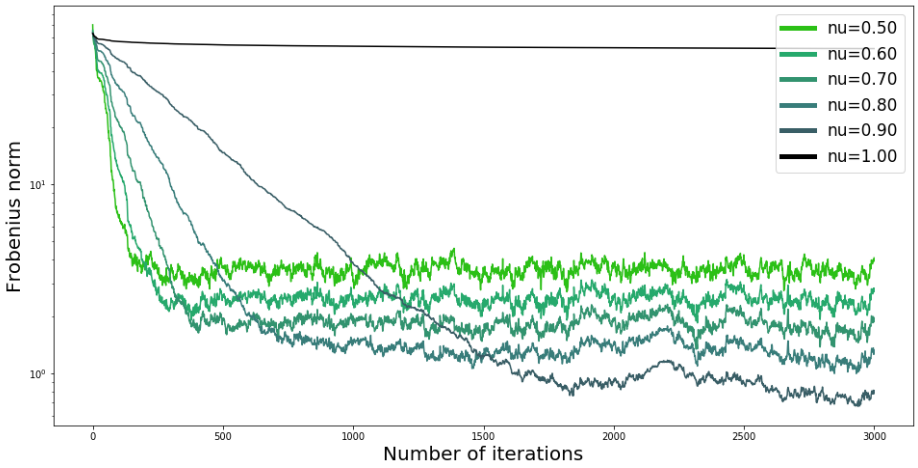

Fig. 3.2: Frobenius norm of the difference between learnt and ground-truth coefficient matrices across time, in log-scale. Darker curves are associated to models with higher forgetting factors.

In figure 3.3, models were all run with a forgetting factor of 0.9 but with different levels of Gaussian noise. As can be reasonably expected, noisy models are harder to learn and yield higher Frobenius norms. Surprisingly, it appears that the relationship between Frobenius norm and noise levels is almost perfectly linear (with some deviation).

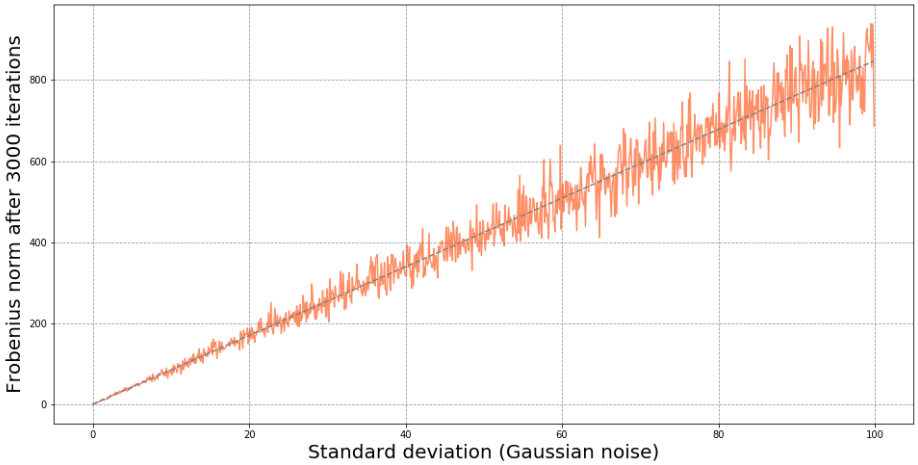

Fig. 3.3: Frobenius norm after 3000 iterations, expressed as a function of the standard deviation of the Gaussian noise term.

Performance is not only affected by the noise level but also the number of input variables, as shown in figure 3.4. However, performance seems to start increasing when the number of explanatory variables exceeds 20.

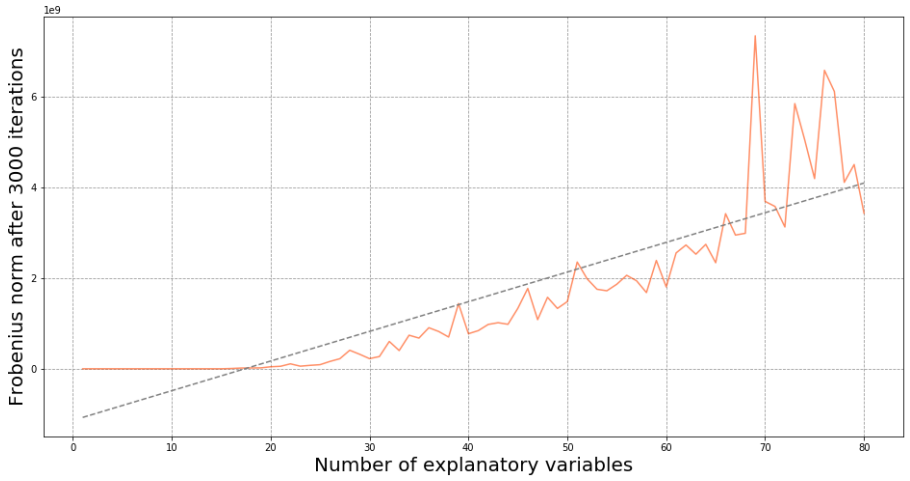

Fig. 3.4: Frobenius norm after 3000 iterations, expressed as a function of the number of explanatory variables in the model.

It is a noteworthy that RLS exhibits extremely fast convergence []. As could be seen in figure 3.1, models with higher average errors have the ability to convergence much faster than the others. This phenomenon is better shown in figure 3.5 below.

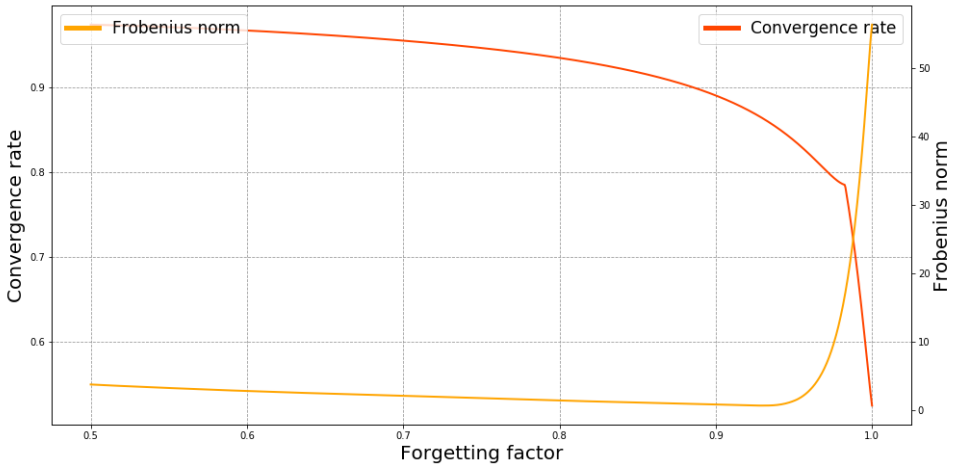

Fig. 3.5: Convergence rates and Frobenius norms as functions of the forgetting factor. Both metrics are based on 3000 simulation steps. Convergence rate is related to the convergence of the prediction error.

Convergence rate has been computed as a function of the excess of error above the global minimum across the whole learning process. The error is normalized and scaled in order

to ensure that the convergence metric is in the range $[0, 1]$.

$$C_r(x) = 1 - \frac{1}{T} \int \frac{x - \min_i x_i}{\max_i x_i - \min_i x_i} dx \tag{3.1}$$

Convergence rate decreases monotonically with the forgetting factor. However, the frobenius norm after 3000 steps does not seem to be monotonic but rather unimodal. In consequence, the value of the forgetting factor that minimizes the Frobenius has been computed using a scalar optimization method. Using the Brent method, a derivative-free technique efficient for finding global optima in unimodal functions [1], the minimum Frobenius norm is reached when the forgetting factor is equal to 0.929.

3.2 scalability

Scalability is the ability of a big data system to process an increasing amount of incoming data by having recourse to an increasing amount of resources. It can be measured by the scaleup, the ratio between the amount of data processed by the model with two different amount of resources but while being run for the same amount of time.

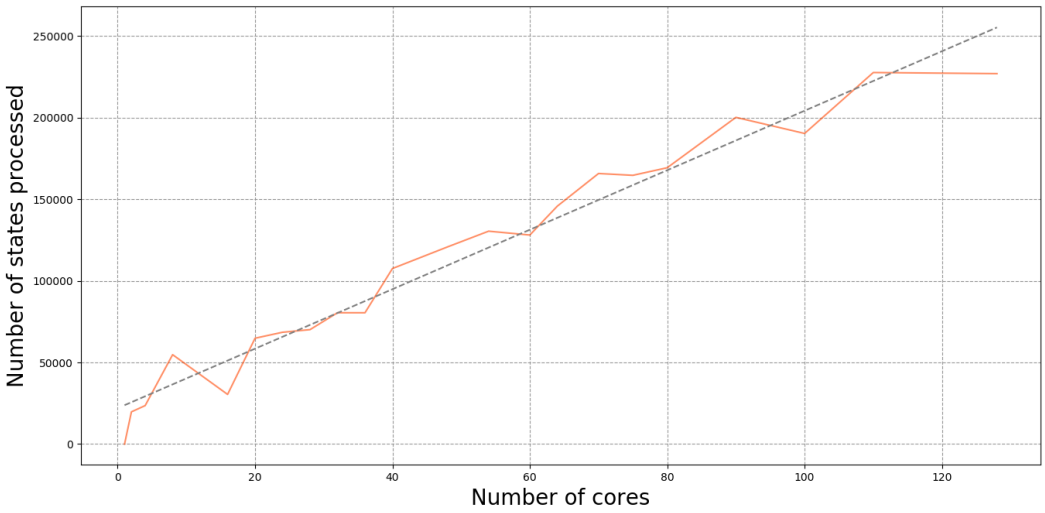

Fig. 3.6: Scalability of the RLS algorithm, measured as the number of states processed in in a fixed amount of time (timeout of 30 seconds), as a function of the number of cores per node in the cluster.

Scalability is measured as the number of states processed by the last map transformation. As can be observed in figure 3.6, the algorithm's scalability benefits from the addition of extra cores. Because of the timeout of 30 seconds, much less states are able to reach the last stage of the pipeline.

4. CONCLUSION

During this project, a distributed version of the RLS algorithm has been implemented using both Kafka distributed messaging server and Kafka streaming API. The algorithm is both scalable with respect to the number of models and output variables. It has also been observed that the speed of convergence is affected by the value of the global optimal, and that higher forgetting factors yield better performance.

As future work, a dynamic forgetting factor [2] could be considered in order to guarantee both high convergence speed and high convergence rate.

BIBLIOGRAPHY

[1] Patricio Basso. “Iterative methods for the localization of the global maximum”. In: *SIAM Journal on Numerical Analysis* 19.4 (1982), pp. 781–792.

[2] Richard M Johnstone et al. “Exponential convergence of recursive least squares with exponential forgetting factor”. In: *Systems & Control Letters* 2.2 (1982), pp. 77–82.

[3] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. 2011, pp. 1–7.

[4] Matei Zaharia et al. “Spark: Cluster computing with working sets.” In: *HotCloud* 10.10-10 (2010), p. 95.