**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

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 producer and retrieved by a Kafka consumer as a Spark DStream. The later is then 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 $\beta_j$ 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.

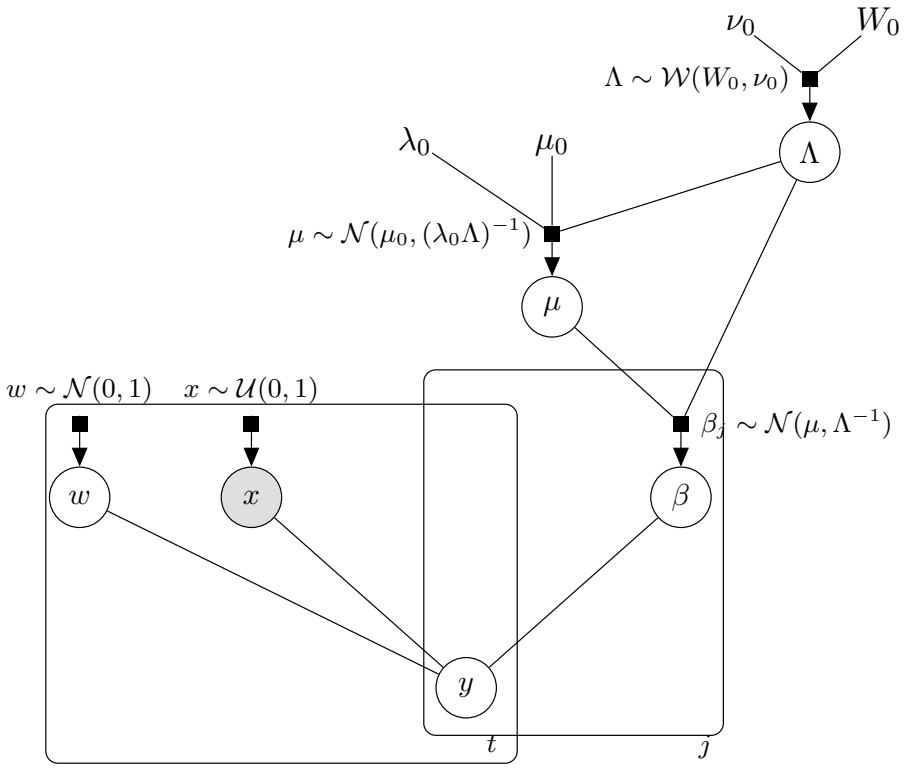


*Fig. 2.1:* Bayesian network representing the randomly generated samples. Plate notation indicates variable repetition across time and output variables, respectively.

As can be observed from the plate notation in figure 2.1, coefficient vectors $\beta_j$ are computed only once for each $j$, and $w$, $x$, $y$ are sampled at each time step $t$, where $y_j = \beta_j x$. Mean vector $\mu$ and precision matrix $\Lambda$ are sampled only once from a normal-

Wishart distribution. Default parameter matrix $W_0$ is the diagonal matrix, prior mean vector $\mu_0$ is the zero vector, chosen scaling parameter is 1, and $\nu_0$ 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 $\beta$ 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

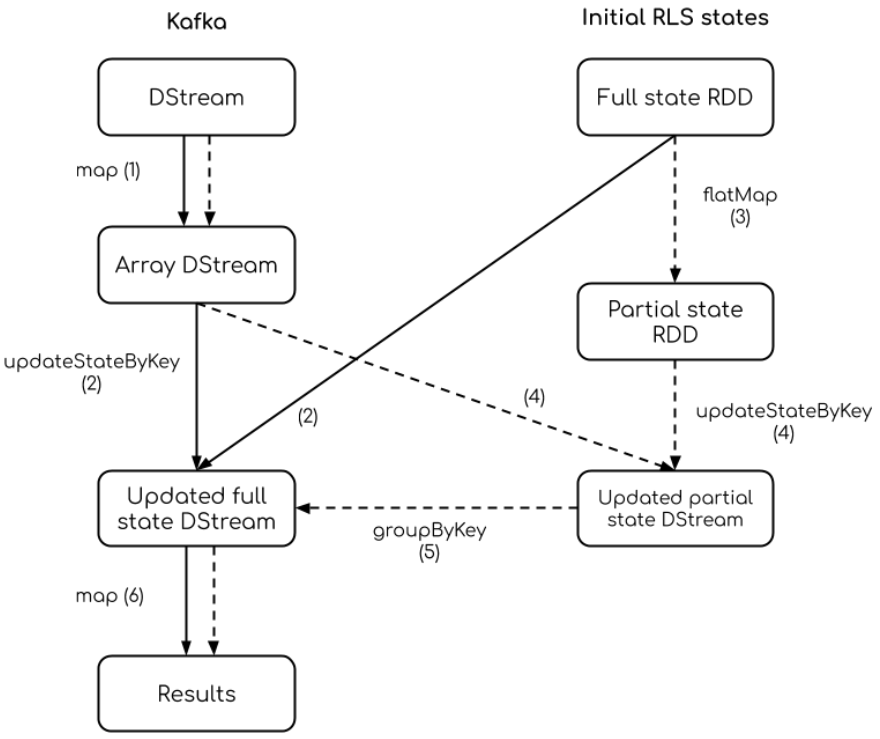


*Fig. 2.2:* Lineage graph for the proposed architecture. Dashed lines indicate the Spark transformations applied in the distributed version and plain lines indicate the transformations applied in the fully-vectorized version.

# 3. RESULTS

## *3.1 Performance*

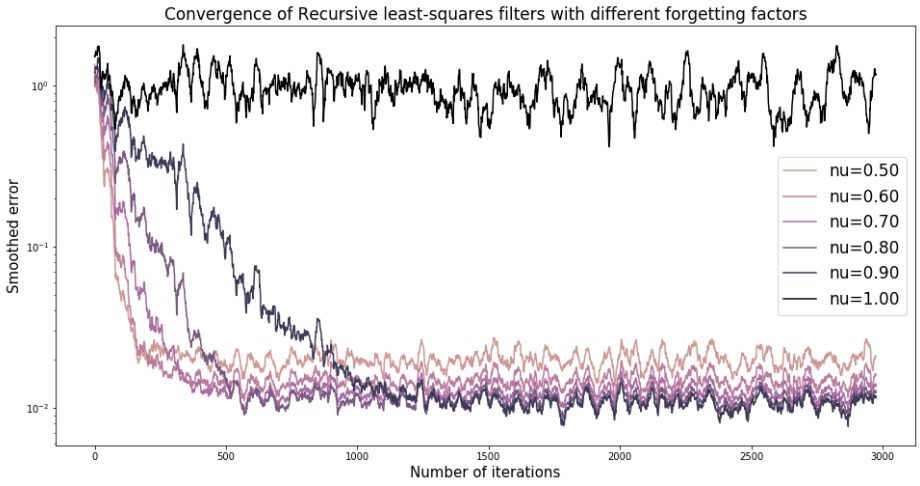


*Fig. 3.1:* Lineage graph for the proposed architecture. Dashed lines indicate the Spark transformations applied in the distributed version and plain lines indicate the transformations applied in the fully-vectorized version.

### *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.

TODO: sub-linear scaleup?

# 4. CONCLUSION