Parallel Computing

Rémy Deshayes and Sarah Lauzeral

February 2021

Please note that this work was originally made in French and has been translated for sharing purposes

We ask our reader to note that this is a very early stage attempt to code in a distributed setting. Results are not particularly striking as we do not distribute on enough variables. Furthermore, the execution times of our distributed algorithms are worryingly unstable.

1 LASSO Regression

1.1 Brief introduction to the LASSO

LASSO is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the resulting statistical model. The method is especially useful when the number of covariates is well above the number of observations.

We recall the Lagrangian form of the LASSO:

$$\min_{\beta \in \mathbb{R}^p} \left\{ \frac{1}{n} ||y - X\beta||_2^2 + \lambda ||\beta||_1 \right\}$$

with y a vector of dependant variables, X the covariate matrix of size
n (number of observations) $\times$ p (number of covariates), β is a weight vector and λ the regularization parameter.

There are many examples of subgradient methods to crack this problem - indeed the LASSO objective function is non-differentiable.
In this project, we chose to focus on the famous Coordinate Descent algorithm by Friedman, Hastie and Tibshirani in *Regularization Paths for Generalized Linear Models via Coordinate Descent* (2010)

1.2 Why is LASSO an interesting candidate in a distributed setting?

Over the years, the amount of available data has sky-rocketed to the extent that some training datasets cannot be stored in a single machine memory.
In that case, traditional subgradient methods such as the Coordinate Descent algorithm can no longer be used directly. Indeed, `Python` and `R` packages often kick off by loading the entire dataset on the RAM.

2 Implementations

2.1 Sanity Checks

To ensure that our functions were working properly we used `Sk-Learn`'s β vectors as a benchmark and cross-checked the values obtained with our function.

Among other usual sanity checks is looking over the "lasso path" i.e the different λ values during the execution.

2.2 First shot

Firstly, we tried to code most of the content from scratch. We created the `Baseline_Lasso` function - inputs are the covariates dataframe, `alpha` which is the λ in the optimization program, the maximum number of iterations allowed in the gradient descent process and finally `intercept`, a Boolean.

2.3 Scikit-learn

Secondly, we used `Sk-Learn` to code our LASSO regression. We assigned the `Lasso` function to an object named `regLasso` - with arguments `fit_intercept` and `normalize`. Then, we fit it to the data in order to find the best βs.

2.4 Code optimization

For the sake of brevity, our LASSO implementation does not feature an intercept.

2.4.1 Cython

We coded 2 algorithms in Cython. For the first one (V1), we coded from scratch various computer algebra functions in C. For the second algorithm (V2), we used the `Blas` library.

V1 - Cython "homemade"

We specified three computer algebra functions in C and a 4th one for the l_1 regularization:

- `loss_c` $= y - y_pred$
- `y_pred_c` $= X_j \times \beta_j$
- `rho_c` $= \sum_{i=1}^{n} X_{i_j} \times y_pred_i$

Those functions are then passed into the C function `fit1` which allows to perform the LASSO gradient descent. This function compiled correctly but failed the β sanity check. Indeed, after wrapping the function in the `Python` function `Cython_Lasso1`, the only output is a bunch of zeros. We reckon the problem must lie in one of the functions round-up.
However, we still kept this implementation to compare running times between this "homemade" version and the V2 `Blas` version.

V2 - Cython and `Blas`

`Blas` gathers various easy-to-use and efficient functions to perform computer algebra. From the library, we used:

- `dcopy(&n, &y[0], &inc, &beta_tmp[0], &inc)`
  to initialize the intercept
- `daxpy(&n, &beta[j], &X[0, j], &inc, &beta_tmp[0], &inc)`
  performs the product $X_{0_j} \times \beta_0$ and compute the intercept
- `ddot(&n, &beta_tmp[0], &inc, &X[0, j], &inc)`
  update β each iteration by computing the product $X_j \times \beta$

As before, we wrap this C function `fit2` in a `Python` function `Cython_Lasso2` that reads C and initialize variables.

2.4.2 Parallel Cython

To distribute our 2 Cython algorithms, we first checked that everything compiled properly when using the `nogil` argument. For this to work fine, the studied function must only include computer algebra and C objects. Furthermore, we took great care in only specifying `Python` objects in the `Python` function wrapping the C code.

In our project, we chose to distribute on the number of covariates - p in the optimization program introduced in section 1 - as this is often seen as the most efficient method (Trofimov and Genkin, *Distributed coordinate descent for L1-regularized logistic regression* 2015).

We use the `cython.parallel` module and the `prange` function which allows to parallelize in the covariates loop.

3 Comparison

We measured the execution time of the different algorithms with the `timeit` package and a function coded by Prof. Dupré - check `http://www.xavierdupre.fr/app/ensae_teaching_cs/helpsphinx3/notebooks/cffi_linear_regression.html`.

XS dataset: 1452 lines & 16 columns - housing dataset. Our target is to use the LASSO regression to predict a property price depending on 15 numerical variables such as the number of bedrooms, the lot size, or the number of cars one can fit in the garage.

XL dataset: 28506 lines & 11 columns - French YouTube dataset. Target is to predict the number of likes for every given video from different variables such as release date.

Model	XS	XL
`Numpy`	4.480	6.810
`SkLearn`	0.005	0.045
`Cython_homemade`	0.016	0.350
`Cython_Blas`	0.002	0.041
`Parallel_Cython_Homemade`	0.015	0.330
`Parallel_Cython_Blas`	0.002	0.045

Table 1: Execution Time in seconds

We notice that `Sk-Learn` is especially efficient. Our implementation from scratch backed by `Numpy` is over a 1000 times slower than the one available on `Sk-Learn`. The "homemade" Cython implementation cuts this gap by a factor 10. Globally, our cython implementation using `Blas` performs well and yields times close to `Sk-Learn` ones.

Comparing results from the XL and XS datasets underlines that our Cython implementation from scratch does not scale up well. On the contrary, the `Sk-Learn` function execution times remains stable and it even performs better than our Cython + `Blas` algorithm.

To conclude, again, we ask our reader to note that this is a very early stage attempt to code in a distributed setting. Results are not particularly striking as we do not distribute on enough variables. Furthermore, the execution times of our distributed algorithms are worryingly unstable.

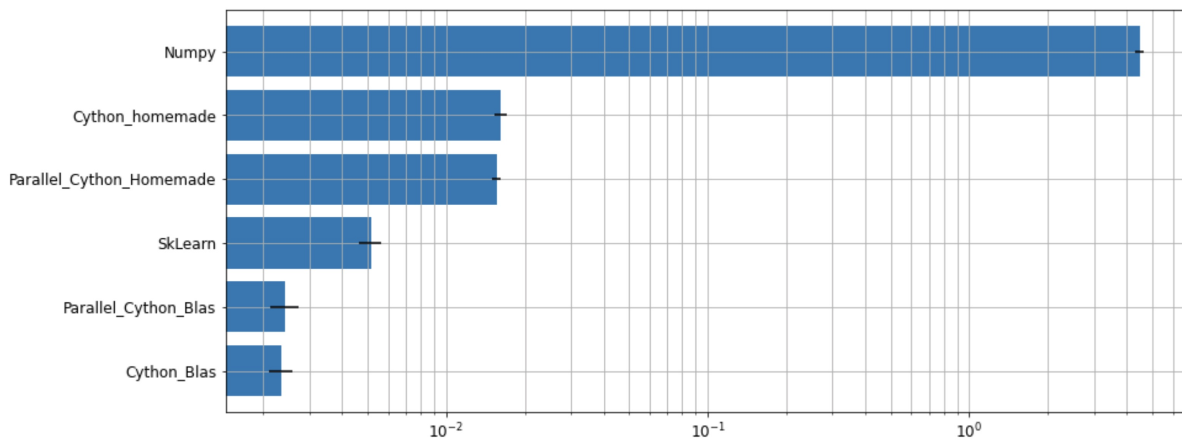

Figure 1: Execution time on the XS dataset

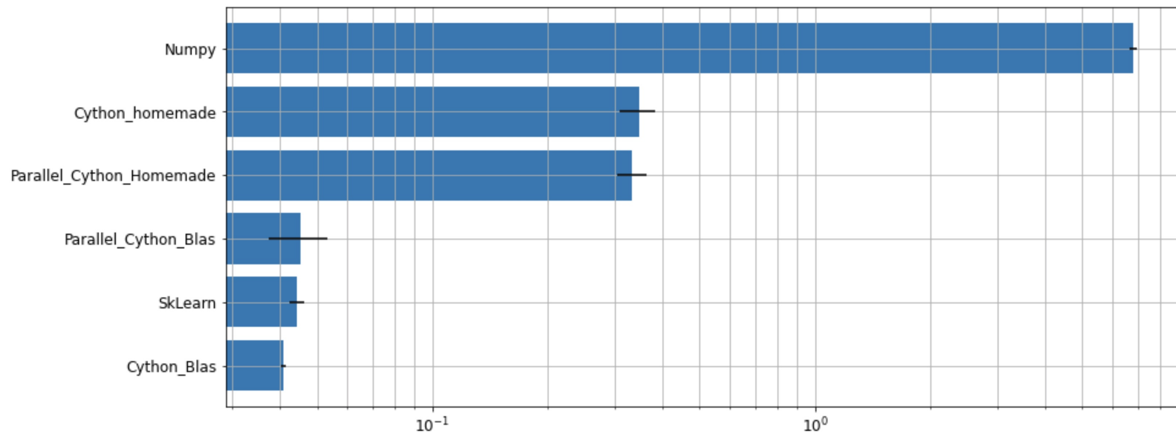

Figure 2: Execution time on the XL dataset - run 1

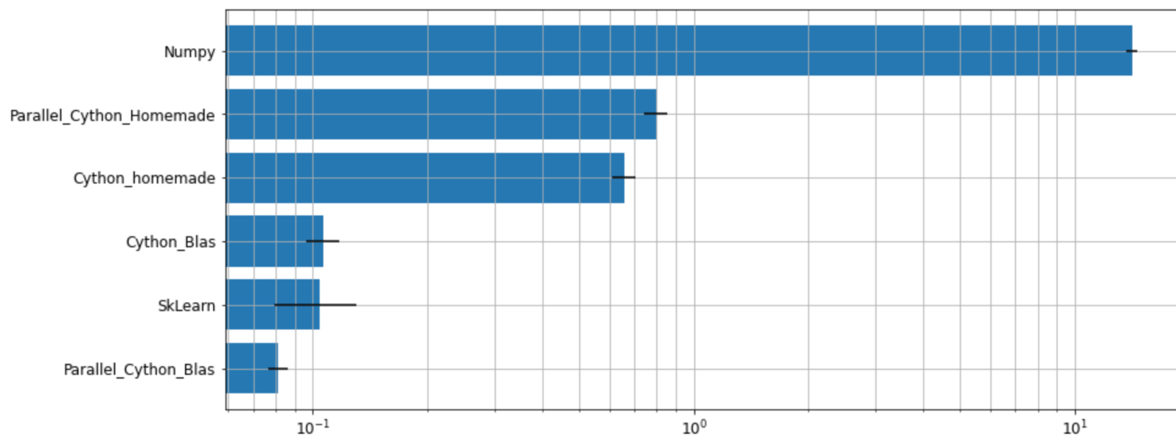

Figure 3: Execution time on the XL dataset - run 2

4 Future work

Working with a larger, LASSO-suited dataset would allow us to back our first opinions.

On another note, we only tried to distribute on the number of covariates, distributing on the number of observations could yield some interesting reference point.

Finally, we chose a very specific method for subgradient descent, choosing another one might yield different conclusions.