Distributed Lasso

CME 323 Project Report

Sebastien DUBOIS, Sebastien LEVY

06/05/2016

Contents

Introduction

The Lasso is a simple linear model used to tackle a wide range of machine learning problems. By adding a L1 penalization to the ordinary least squares, it induces sparsity in the coefficients, making the algorithm very efficient even when the number of features is bigger than the number of observations. Another interesting characteristic of the Lasso is its piece-wise linear coefficient path, which can be leveraged for efficient computations. Although simple gradient methods can be applied to solve the underlying convex minimization problem, there exists a couple of exact methods which are as efficient, if not faster. Such methods are usually preferred in sequential frameworks to guarantee sparsity and compute the whole coefficient path.

In this project, we propose a distributed algorithm to solve the Lasso based on Least Angle Regression (LARS). Our algorithm guarantees the sparsity of the solution, can handle all types of distributed matrices, and computes the whole coefficient path. We show that the complexity of the proposed algorithm is promising, since the communication cost is comparable to gradient methods, while providing a useful parallelization of the sequential version (similar amount of total work, logarithmic depth in the large dimension).

This report is organized as follows. First, we briefly present the Lasso and different methods to solve it. Then, we discuss various challenges in its parallelization and review existing attempts to solve the Lasso in distributed frameworks. This leads us to focus on LARS, which algorithm is detailed in section 3. We finally propose a distributed version of LARS (D-LARS) for three types of data matrices: 'Short and Fat' (section 4), 'Tall and Skinny' (section 5), and 'Almost Square' (section 6). In each case, we provide pseudo-code to reflect the challenges of the implementation, and analyze the complexity in terms of communication cost and computation time. We conclude on why we think the proposed method is promising by comparing it with existing solutions, especially SGD-based implementations.

1 Lasso Regression

1.1 Definition, Penalization and Sparsity

The Lasso regression is a linear regression with a L1 penalization. The penalization term added to the ordinary least squares creates bias but decreases the variance. Since the mean square error (the quantity to minimize) is the sum of the squared bias and variance, we can often achieve better accuracy by optimizing the regularization coefficient to reach optimal trade-off.

Compared to other types of penalization, the main advantage of the L1 norm is that it combines two great advantages, sparsity and convexity. While staying convex and guaranteeing a unique minimum, it induces sparsity in the coefficients, i.e. a lot of the coefficients are zero. This is a real advantage for interpretability, computing time and storage memory. This type of penalization is also well adapted to cases where $p > n$. Indeed, there would be an infinite number of solutions without penalization, but the L1 norm insures uniqueness of the solution, with at most n predictors in the model (those with non-zero coefficient).

The minimization problem to solve is the following:

$$argmin_\beta \sum_{i=1}^{n} (X_i^T \beta - y_i)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

The λ coefficient is the regularization parameter. By changing its value we can find the coefficient path (the values of each coefficient for every value of λ). It has the particularity to be piece-wise linear for the Lasso, and slopes change only when a new variable enters the set of active variables (those with non-zero coefficient).

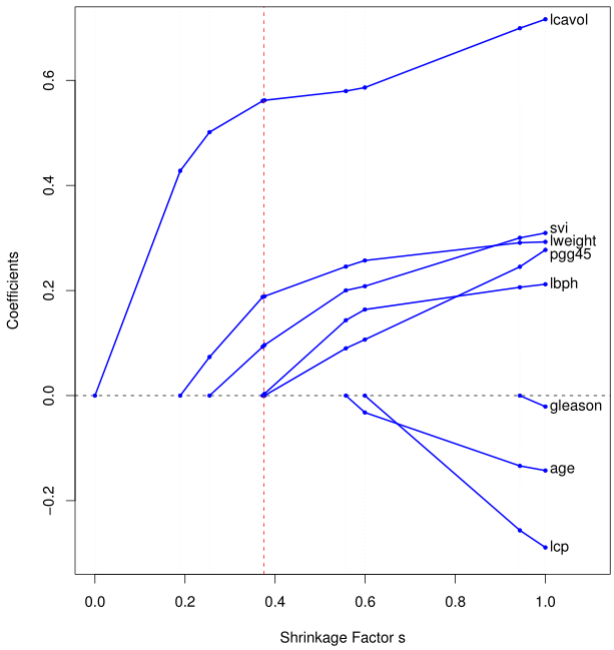

Figure 1: Example of a Lasso coefficient path (Figure 3.10 in [5])

Besides providing insight on the model and the different importance of features, it can also ease the search for optimal λ by cross-validation. With the path, we have all the coefficients for every value of λ.

The interest of studying Lasso extends to other interesting methods. By adding, in addition to the L1 penalty, an L2 penalty we get the Elastic Net. This version combines the advantages of Lasso but handles correlated features better. The L1 penalty can also be added to more complicated methods in a similar way. The most popular are logistic regression, nearest neighbors, LDA and SVM. It is also used in boosting methods.

1.2 Sequential Solving Methods

1.2.1 Gradient Methods

The simplest way to solve the optimization problem is to simply use any general gradient based solving method. The convexity of the loss function guarantee to find the global minimum. Two methods are mostly used to solve this problem: gradient descent and stochastic gradient descent (SGD). Because the penalty is not differentiable in 0, equivalent subgradient methods are used.

At each step, the gradient descent method simply computes the gradient of the quantity to minimize and moves of a step η in the other direction. This will converge quite quickly:

to get a precision ϵ, we will need only $O\left(\log\left(\frac{1}{\epsilon}\right)\right)$ iterations. However, at each step we will need to see all the observations making the iterations long, and somehow useless in the beginning where an accurate estimation of the best direction to go is not essential.

Another alternative, is stochastic gradient descent (SGD) in which we use the fact that the minimizing function can be written as a sum of differentiable function (except in 0). We will then compute the gradient of all the elements of the sum and update them one after the other. At each pass (all the observations give their update), we shuffle the data. This methods has much smaller iterations but need $O\left(\frac{1}{\epsilon}\right)$ iterations to converge with a precision ϵ.

The main advantage of those methods are that they can give a good approximation of the solution quite rapidly. The methods are really simple, only need the gradient (no order 2 differentials) and are very general. However, on the particular optimization problem of Lasso, approximating the solution tends to lose the sparsity [7], the solutions won't have many zero coefficients but small values instead. Although we could hard threshold the coefficients, we would still get more computations and more memory needed to store all the non-zero coefficients. The method is also less adapted to compute the whole coefficient path. It can use solutions from a close value of λ as a warm start but would still need to solve many optimization problems and won't take advantage of the piece-wise linear property of the path.

1.2.2 Coordinate Descent

Coordinate Descent [3] is the most recent of the successful methods proposed to compute the Lasso path (and actually to solve any Elastic-Net). This algorithm is known to be very fast, and is the one used in the well-known R package `glmnet`.

The basic idea behind coordinate descent is to start with an estimate $\tilde{\beta}$ and update each of its entries one by one, using the gradient of the objective function (loss + L1 penalty) considering all but one entries fixed.

In practice, the coordinate descent algorithm starts with $\tilde{\beta} = 0$ (corresponding to an infinite penalty $\lambda = \infty$), and slightly decreases the value of the regularization parameter λ after a solution is found.

This technique has proved to be very efficient because it leverages the fact that most of the coefficients are zero as well as warm starts, while computing the whole coefficient path. Therefore, the algorithm must do a lot of iterations (for many different values of λ) but each of them requires only a few step. This technique also guarantees sparsity in the coefficients.

1.2.3 Least Angle Regression

Least Angle Regression (LARS) is a solving technique based on the computation of the coefficient path, taking advantage of its piece-wise linear property. It was originally proposed by Efron et al. [2] as a new forward selection method. The coefficient path it computes was found out to be very similar to the Lasso path. The path is actually the *exact* same when no coefficient crosses zero in the path. By slightly modifying the algorithm (see section 3.5), the exact Lasso solution can be computed in any cases. Before the emergence of coordinate descent, it was the main technique to solve this problem.

The algorithms starts with all coefficients to zero, it finds the predictor most correlated with the residual, and increases its coefficient until another predictor is equally correlated with the residual. Then, it increases both coefficients in a way that keeps the correlations tied, until another predictor is equally correlated with the residual, *etc* ... This can be

interpreted as finding the *least angle* to increase the coefficients. The algorithm is detailed in section 3.

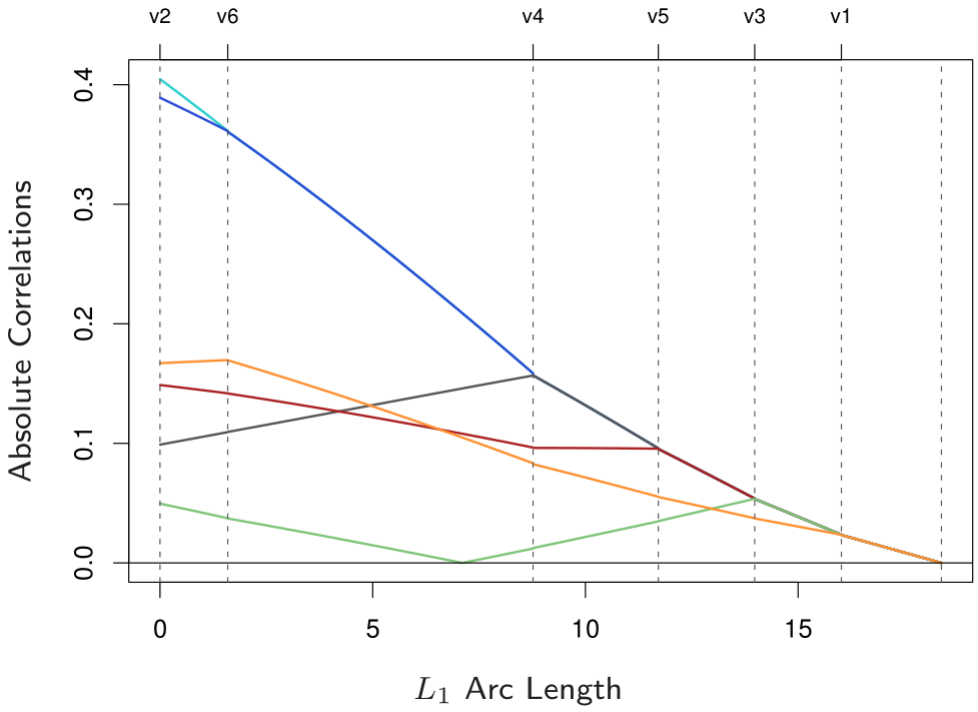

Figure 2: Progression of the absolute correlations during each step of the LARS procedure, using a simulated data set with six predictors. The labels at the top of the plot indicate which variables enter the active set at each step (Figure 3.14 in [5]).

LARS computes the whole coefficient path, but interestingly its complexity is the same as the resolution of an ordinary least squares. It is also worth noting that the algorithm performs $\min(n, p)$ steps only.

2 Parallelization

2.1 Objectives

In this project we will focus on designing a distributed solving algorithm for the Lasso. We will set 4 objectives for our technique:

1. **Reasonable complexity** : We want to limit the computation time, thus we aim a total work comparable to sequential methods and a logarithmic depth. We also want the communication cost to stay close to those required by other existing methods, such as Stochastic Gradient Descent for the $n \gg p$ case (small dimension $\times$ number of machines $\times$ number of iterations).

2. **Guarantee sparsity** : The sparsity in the coefficients is one of the major interests of the Lasso. For very big models ($p \gg n$ or $p \sim n$), we can easily assume that the number of relevant predictors should remain small. Regarding the complexity of the algorithm, sparsity has also two advantages:

   - having many zero values largely decrease the storage necessary
   - in a distributed framework, we will need to communicate the coefficient estimate between different machines, therefore its sparsity will help controlling the communication required by the algorithm.

   To guarantee sparsity, it would then be interesting to focus on *exact* solving methods.

3. **Whole coefficient path** : As explained before, an interesting feature of the Lasso method is its piece-wise linear coefficient path. Various sequential solving methods can output the whole path for a similar computational cost (LARS, Coordinate Descent). This path can be used to grasp a better understanding of the model studied and to get the optimal penalization coefficient (λ) by cross validation without computation overhead.

4. **All types of matrices** : An important advantage of the Lasso is its ability to deal with problems where $p \sim n$ or $p \gg n$. Indeed, the solution is unique thanks to the regularization, and the solution's sparsity guarantees interpretability. The last case, $n \gg p$, is very common in machine learning problem, especially in medicine or biology. Thus, we find it really valuable to design algorithms which can deal with the three types of data matrices outlined previously.

We will show in the next parts that the existing methods do not satisfy those 4 objectives. This justify the development of a new distributed method based on Least Angle Regression.

2.2 Existing methods

In the distributed framework, only two methods are really implemented to solve the Lasso:

2.2.1 Using SGD (Spark)

On Spark, like various other regression and classification methods, the Lasso is solved using stochastic gradient descent. To avoid shuffling the data, which would imply a large communication cost ($O(np)$), distributed sampling is used on the observations at each iteration. A subgradient is used to deal with non differentiability of the L1 norm in 0.

This method does not satisfy 2 of our 4 objectives. By computing approximate solutions, we lose the sparsity in the coefficients [7]. Like the sequential version, it does not compute the full coefficient path, and although warm starts could be used, we would need to solve a large number of optimization problems to get it which would involve unreasonable complexities. Moreover, this method is only really adapted to the $n \gg p$ case. It is implemented for matrices distributed by rows and the big advantage of SGD is that it can stop before seeing all the observations.

Because it fails at most of our objectives, we decide to discard this method for this project and to explore distributed version of other solving methods.

2.2.2 Shotgun (Distributed Coordinate Descent)

The Shotgun method developed by Bradley et al [1] is a distributed version of the shooting method (Coordinate Descent). The general idea is to update the coefficients as in the sequential coordinate descent, but in parallel. The resulting method works well with uncorrelated features but updates can conflict when correlations are too large.

It has been proved that the number of machines on which it can be parallelized without diverging depends on the biggest eigenvalue of X^TX: ρ (at most $\frac{p}{\rho}+1$). This is problematic since this quantity would be too costly to compute in general (note that X^TX do not fit in memory when $p \gg n$). In addition, when ρ is too large, we would only be able too parallelize the algorithm by making several machines work on the same update. But this would yield important communication cost and suboptimal algorithm. Other techniques to avoid divergence would be to lock when performing an important update but that would go against our distributed framework.

The need to solve the optimization problem for a large set of values of λ would also involve unreasonable communication cost. Another disadvantage of this method is that it is only adapted to matrices stored by columns. When $n \gg p$ and if it is stored by rows, once again, there will be communication issues.

Finally, Zeng et al. compared the performances of the Shotgun algorithm on different machine learning platforms [9]. They report poor results on Spark compared to the other frameworks, which let us think this method is not well suited for Spark (which is our targeted framework). In addition, they claim this was due to an excess of communication (even though they study this algorithm on Short and Fat matrices $p \gg n$).

It is clear that this algorithms does not satisfy our objectives. Moreover, its convergence to the solution depends a lot on the data, which is not something desirable.

2.3 What about Least Angle Regression ?

In section 1.2 we presented the three most famous methods to solve the Lasso sequentially. Attempts were made to design a distributed version of the first two, but as described above we found they had several drawbacks. Thus, it seems reasonable to look at the third option: Leas Angle Regression. By essence, it guarantees the sparsity and computes the whole path. We also know that the computation of the path does not add any overhead in the sequential framework.

Another main advantage is that it is based on the covariance matrix of the active variables, whose size cannot exceed the smallest dimension ($\min(n, p)$). It is therefore well adapted to cases where $n \gg p$ and $n \ll p$. The number of iterations of the algorithm is also quite small in those cases ($\min(n, p)$). Most of the operations appear embarrassingly parallel (finding the next feature to enter the model, normalizing the data, updating the coefficients) or easily doable on the driver (solving linear system). We will see that we can also deal with almost square matrices with either an incomplete path, by approximation, or by increasing the communication cost.

In the next part, we will study the full sequential algorithm and we will then study separately the distributed algorithm for each one of the three types of matrices.

3 Sequential LARS algorithm

3.1 General idea

The LARS method was introduced in 1.2.3. In practice, the two main parts of the algorithm are the computations, at each iteration, of:

- the new coefficient direction δ_k that keeps tied all the correlations of active variables with the residual. This parts solves a linear system of size k (the iteration number)
- the step $\alpha_{k+1} \in [0, 1]$ to go on the computed direction at this iteration in order to update the active coefficients, $\beta_{k+1} \leftarrow \beta_k + \alpha_{k+1}\delta_k$

3.2 Algorithm

Algorithm 1 Sequential Least Angle Regression Algorithm

procedure LARS(X, y) ▷ X n x m matrix, y vector of size n
    Normalize(X)

    $r_1 \leftarrow y - \bar{y}$
    $A_1 \leftarrow \{argmax_j(|X_j^T r_1|)\}$
    $\beta_1 \leftarrow 0$

    for k = 1 to m-1 **do**
        $\delta_k \leftarrow (X_{A_k}^T X_{A_k})^{-1} X_{A_k}^T r_k$

        Let $j_A \in A_k$
        for $j \in \bar{A}_k$ **do**
            $\alpha_j^- \leftarrow \frac{|X_{j_A}^T r_k| - X_j^T r_k}{|X_{j_A}^T r_k| - X_j^T X_{A_k} \delta_k}$
            $\alpha_j \leftarrow \frac{|X_{j_A}^T r_k| + X_j^T r_k}{|X_{j_A}^T r_k| + X_j^T X_{A_k} \delta_k}$
            if $\min(\alpha_j^+, \alpha_j^-) < 0$ **then**
                $\alpha_j \leftarrow \max(\alpha_j^+, \alpha_j^-)$
            else
                $\alpha_j \leftarrow \min(\alpha_j^+, \alpha_j^-)$
            end if
        end for
        $j_{k+1} \leftarrow argmin_{j \in \bar{A}_k} \alpha_j$

        $A_{k+1} \leftarrow A_k \cup \{j_{k+1}\}$
        $r_{k+1} \leftarrow r_k - \alpha_{j_{k+1}} X_{A_k} \delta_k$
        $\beta_{k+1} \leftarrow \beta_k + \alpha_{j_{k+1}} \delta_k$
    end for
return coefs
end procedure

3.3 Proof

We prove in this section why the algorithm above does compute the LARS solution.
Using

$$r_k(\alpha) = r_k - \alpha X_{A_k}\delta_k$$

we get:

$$X_{A_k} r_k(\alpha) = X_{A_k}^T r_k - \alpha X_{A_k}^T X_{A_k}\delta_k = (1-\alpha)X_{A_k}^T r_k$$

which shows that δ_k is indeed the direction for which all the active variables keep identical correlation with the residual.
Also $\forall j_A \in A_k$, the correlation is:

$$Corr(j_A, r_k(\alpha)) = (1-\alpha)X_{j_A}^T r_k$$

and for any other predictor j :

$$Corr(j, r_k(\alpha)) = X_j^T r_k - \alpha X_j^T X_{A_k}\delta_k$$

For each predictor $j \in \bar{A}_k$, we are looking for the value of α_j such that:

$$|Corr(j, r_k(\alpha))| = |Corr(j_A, r_k(\alpha))|$$

If $Corr(j, r_k(\alpha)) > 0$, which is equivalent to $X_j^T r_k > \alpha X_j^T X_{A_k}\delta_k$, then

$$\alpha_j^- = \frac{|X_{j_A}^T r_k| - X_j^T r_k}{|X_{j_A}^T r_k| - X_j^T X_{A_k}\delta_k}$$

Otherwise:

$$\alpha_j^+ = \frac{|X_{j_A}^T r_k| + X_j^T r_k}{|X_{j_A}^T r_k| + X_j^T X_{A_k}\delta_k}$$

We can then simplify this expression by taking

$$\alpha_j = \begin{cases} \min(\alpha_j^+, \alpha_j^-) & \texttt{if } \min(\alpha_j^+, \alpha_j^-) \geq 0 \\ \max(\alpha_j^+, \alpha_j^-) & \texttt{if } \min(\alpha_j^+, \alpha_j^-) < 0 \end{cases}$$

3.4 Matrix inversion and Cholesky decomposition

As outlined in the algorithm above, at each step we need to compute the new coefficient direction $\delta_k \leftarrow (X_{A_k}^T X_{A_k})^{-1} X_{A_k}^T r_k$. This means that we need to solve the linear system $C_k x = X_{A_k}^T r_k$ where C_k is the covariance matrix between active variables (of size $k \times k$).

A classic method to solve such an equation is to compute the LU decomposition of C_k (in time $O(k^3)$), to then solve two triangular systems (in time $O(k^2)$).

For the LARS algorithm, we can actually do a lot better. First notice that C_k is symmetric so we actually compute the Cholesky factor G_k such that $C_k = G_k G_k^T$. Also, at each step C_k is extended by one last column/row, so the first $k \times k$ block of G_{k+1} is actually G_k. Hence we only need to compute the last new row of G_{k+1} at step $k+1$, which is done in k^2 only (compared to k^3 to compute the whole factor). A few more details on the exact computation of the Cholesky factor are given in section 6.2.1.

Therefore, from the first to the last iteration, we store the current Cholesky decomposition of the covariance matrix $X_{A_k}^T X_{A_k}$. It is updated at each step in $O(k^2)$ time, and used to solve the linear system, also in in $O(k^2)$ time, giving δ_k.

This remark decreases the total computations needed for $\delta_1, \ldots, \delta_I$ by a factor I: from $O(I^4)$ to $O(I^3)$ (I being the total number of iterations, so typically $I = \min(n, p)$).

3.5 Lasso path from LARS?

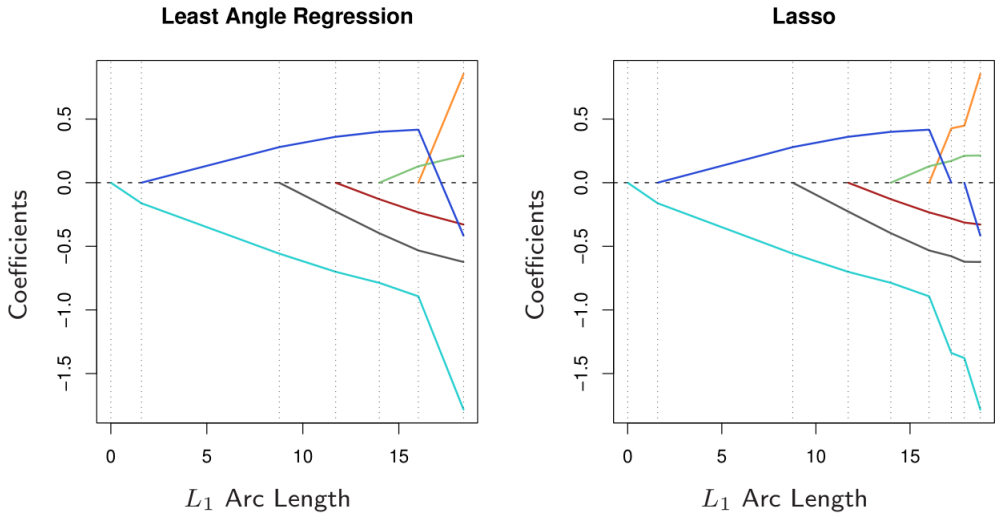

Figure 3: Left panel shows the LARS coefficient profiles on the simulated data, as a function of the L1 arc length. The right panel shows the Lasso profile. They are identical until the dark-blue coefficient crosses zero at an arc length of about 18 (Figure 3.15 in [5]).

Figure 3 shows a comparison of the coefficient paths found by LARS and the Lasso on the same data. We observe that the profiles are identical until one coefficient crosses zero. This is a general result which leads to a simple modification of the LARS algorithm to compute the Lasso path:

> If a non-zero coefficient hits zero, drop its variable from the active set of variables and recompute the current joint least squares direction.

In other words, if the value found for α_k is such that the coefficient of an active variable changes sign, we instead choose α_k such that this coefficient is 0, and then continue the algorithm removing this variable from the active set. In practice, this would be implemented by finding an α_k for each value of the active set at which its coefficient becomes zero [1].

Therefore the LAR(lasso) procedure can have more than $\min(n, p)$ steps but will still be close to this number, since such an event does not happen often in practice. So this method can compute the whole Lasso coefficient path in the same order of computation as that of an ordinary least squares fit.

4 D-LARS for 'Short and Fat' matrices ($n \ll p$)

4.1 Set up

For short and fat matrices, we assume that $n \ll p$, that n^2 fits in memory but p doesn't. Therefore, we must store the data by columns with each of the B machines containing m

[1] If the absolute value of the coefficient is increasing, we just assign a value greater than 1

columns – We assume mn fits in memory ($m \times B = p$). We want our algorithm to

- Scale on p, the large dimension
- Avoid All-to-All communications
- Not broadcast more than a constant number of vector of size n at each iteration to every machine

Because n^2 fits in memory and $I \leq n$, we will store the covariance matrix on the driver. To be able to easily solve the linear system giving the direction $\delta_k \leftarrow (X_{A_k}^T X_{A_k})^{-1} X_{A_k}^T r_k$, we store the Cholesky decomposition of the covariance matrix on the driver, and update it at each iteration. We will also store the current correlation with the residual and the coefficients of previous iterations on the driver. Each machine will only store its m columns.

4.2 Algorithm

When the data is distributed by columns, we use the following algorithm:

Algorithm 2 Short and Fat - Distributed Least Angle Regression

procedure D-LARS(X, y) ▷ X n x p matrix, y vector of size n

Normalize(X) in parallel ▷ Map

$r_1 \leftarrow y - \bar{y}$
Broadcast r_1 to every machine
for j in 1 to p **do**
$\tilde{s}_j \leftarrow 2 * 1_{X_j^T r_1 > 0} - 1$
Emit $(j, |X_j^T r_1|, \tilde{s}_j)$ ▷ Map
end for
$j_1, cor_1, s_1 \leftarrow \max(j, |X_j^T r_1|, \tilde{s}_j)$ ▷ Reduce

Init(j_1, X_{j_1})

for k = 1 to n-1 **do**

Broadcast r_k to every machine
$\delta_k \leftarrow$ **ComputeDelta**(cor_k, X_{A_k}, s)
$j_{k+1}, \alpha_{k+1}, s_{j_{k+1}} \leftarrow min_{j \in \bar{A}_k}$**EmitAlpha**$(\delta_k, cor_k, \bar{A}_k)$ ▷ Reduce

$A_{k+1} \leftarrow A_k \cup \{j_{k+1}\}$
$cor_{k+1} \leftarrow (1 - \alpha_{k+1}) cor_k$
UpdateEstimates$(j_{k+1}, \alpha_{k+1}, X_{A_k}, X_{k+1})$

end for
return β
end procedure

The pseudo-code for **Init**, **ComputeDelta**, **EmitAlpha** and **UpdateEstimates** is provided below. Note that since $n \ll p$, the Cholesky factor as well as the $n \times n$ matrix

containing the coefficients of the path are stored on the driver. The computations of δ_k and $A_k\delta_k$ are also done on the driver.

Algorithm 3 Short and Fat Init

procedure INIT(j_1, X_{j_1})
    $A_1 \leftarrow \{j_1\}$
    $G_1 \leftarrow \left(X_{j_1}^T X_{j_1}\right)$
    $\beta \leftarrow$ SparseMatrix of zeros $(n \times n)$
end procedure

Algorithm 4 Short and Fat ComputeDelta

procedure COMPUTEDELTA(cor_k, X_{A_k}, s_{A_k})
    ▷ Locally
Compute the new column of the covariance matrix $X_{A_k}^T X_{A_k}$ from dot product of new active variable with previously selected ones
Extend the Cholesky factor G_k with new row, so that $G_k G_k^T = X_{A_k}^T X_{A_k}$

    $c_k \leftarrow cor_k * s_{A_k}$ ▷ $X_{A_k}^T r_k$
    $\delta_k \leftarrow solve(G_k G_k^T x = c_k)$ ▷ Locally
return δ_k
end procedure

Algorithm 5 Short and Fat EmitAlpha

procedure EMITALPHA($\delta_k, cor_k, \bar{A}_k, X_{A_k}$)
    Compute and Broadcast $\tilde{X}_k \leftarrow X_{A_k}\delta_k$ ▷ Map + n Reduce

    for $j \in \bar{A}_k$ **do**
        $\alpha_j^- \leftarrow \left(cor_k - X_j^T r_k\right)\left(cor_k - X_j^T \tilde{X}_k\right)^{-1}$
        $\alpha_j^+ \leftarrow \left(cor_k + X_j^T r_k\right)\left(cor_k + X_j^T \tilde{X}_k\right)^{-1}$
        if $\min(\alpha_j^+, \alpha_j^-) < 0$ **then**
            $\alpha_j \leftarrow \max(\alpha_j^+, \alpha_j^-)$
        else
            $\alpha_j \leftarrow \min(\alpha_j^+, \alpha_j^-)$
        end if
        if $\alpha_j = \alpha_j^+$ **then**
            $\tilde{s}_j \leftarrow +1$
        else
            $\tilde{s}_j \leftarrow -1$
        end if
        Emit $(j, \alpha_j, \tilde{s}_j)$ ▷ Map
    end for
end procedure

Algorithm 6 Short and Fat UpdateEstimates

procedure UPDATEESTIMATES($j_{k+1}, \alpha_{k+1}, X_{A_k}, X_{j_{k+1}}$)
// Update Cholesky factor
    Broadcast $X_{j_{k+1}}$ to A_k
    for $j \in A_k$ **do**
        Emit $(j, X_j^T X_{j_{k+1}})$ ▷ Map
    end for
    $Cov_{:,k+1} \leftarrow \begin{pmatrix} X_{A_k}^T X_{j_{k+1}} \\ 1 \end{pmatrix}$ new column of the covariance matrix
    $G_{k+1} \leftarrow$ Update Cholesky factor G_k ▷ Locally

// Update residual
    $r_{k+1} \leftarrow r_k - \alpha_{k+1} \tilde{X}_k$ ▷ Locally

// Update coefficients
    for $j \in A_k$ **do**
        $\beta_{k+1,j} \leftarrow \beta_{k,j} + \alpha_{k+1} \delta_{k,j}$ ▷ Locally
    end for
end procedure

4.3 Spark Implementation

We implemented the algorithm outlined above in Scala / Spark on Databricks. A notebook with full code and examples can be found at `http://bit.do/dlars_databricks`. In addition, an extract of the core of the code is given in Appendix.

For simplicity, and due to the short time frame of this project, we did not use the Cholesky factor but rather stored the covariance matrix. The linear system is solved at each step with `breeze.linalg`. However we still implemented the smart update of the covariance matrix, so this does not impact the communication, but only the computation time on the driver.

We tested our code on two data sets:

- The `lpsa.data` set provided in Spark MLlib [2]
- A crime data set [3] from the UCI Machine Learning Repository [6].

In both cases we compared our results with the LARS implementation in R, and found identical coefficient paths.

These data sets both have a rather small number of features and observations, and are only intended to highlight the correctness of the implementation. Indeed, our Databricks account was not well fitted to handle very large amount of data. This is also why we do not report execution times or experimentally compare it with other implementations such has `LassoWithSGD` implemented in Spark.

[2] data can be found at `https://github.com/apache/spark/blob/master/data/mllib/ridge-data/lpsa.data`

[3] information can be found at `http://archive.ics.uci.edu/ml/datasets/Communities+and+Crime+Unnormalized`

4.4 Complexity

Let's suppose we only have B machines, each one containing m columns ($m \times B = p$). The complexity analysis is then:

- Normalizing in parallel and computing $y - \bar{y}$ takes $T = O(mn)$.
- Broadcasting r_1 to every machine takes communication $C = O(nB)$ and $T = O(n \log(B))$.
- Finding the maximum correlation takes $T = O(\log(B) + mn)$ time and $C = O(B)$ communication. We first compute the correlation in each machine ($O(mn)$), combine within each machine by taking the max ($O(m)$) and send those values ($T = O(\log(B))$ and $C = O(B)$)
- The **Init** takes $T = O(1)$ time.

- For the k^{th} loop:
  - the broadcast takes $C = O(nB)$ communication and $T = O(n \log(B))$ time
  - **ComputeDelta** is done locally and takes time $T = O(k^2)$ since the linear system is solved using the Cholesky decomposition of the covariance matrix (as detailed in section 3.4).
  - **EmitAlpha** takes $T = O(kn)$ to compute $\tilde{X}_k$, $T = O(n \log(B))$ to broadcast it, $T = O(mn)$ to compute $X_j^T r_k$ and $X_j^T \tilde{X}_k$, $T = O(m)$ to take the minimum within each machine, and $T = O(\log(B))$ to emit each α_j. This gives a total time of $T = O(kn + n \log(B) + mn)$. The communication is $C = O(nB)$ due to the first broadcast.
  - Taking the minimum of the α_j sent by each machine, by doing a reduce, takes $T = O(\log(B))$ and $C = O(B)$.
  - Updating A_k, r_k and cor_k takes $T = O(1)$ because $X_{A_k} \delta_k$ is already computed.
  - **UpdateCholesky** takes communication $C = O(\min(k, B))$ and time $T = O(\min(k, B) + k^2 + \min(k, m)n) = O(k^2 + \min(k, m)n)$ to compute the correlations on each machine ($\min(k, m)n$), send them to the driver ($\min(k, B)$) and update the Cholesky decomposition (see section 3.4).

This gives asymptotically (O are omitted):

$$\begin{aligned} T &= mn + n \log(B) + \sum_{k=1}^{I} \left(n \log(B) + k^2 + nk + mn \right) \\ &= mnI + n \log(B) I + I^3 + nI^2 \end{aligned}$$

$$\begin{cases} T = O(mnI + n \log(B) I + I^3 + nI^2) \\ C = O\left(nB + \sum_{k=1}^{I} nB \right) = O(nBI) \end{cases}$$

4.5 Discussion

Let's first look at the complexity in different cases:

- In the case where we want the full path, we have $I = n$ and then:

$$\begin{cases} T = O(mn^2 + n^2 \log(B) + n^3) = O\left(\frac{pn^2}{B} + n^2 \log(B) + n^3\right) \\ C = O(n^2 B) \end{cases}$$

- With $B = p$ (a machine for each column) we get:

$$\begin{cases} T = O(n \log(p) I + I^3 + nI^2) \\ C = O(npI) \end{cases}$$

- With $B = p$ and $I = n$:

$$\begin{cases} T = O\left(n^2 \log(p) + n^3\right) \\ C = O(n^2 p) \end{cases}$$

We can see that the communication cost is the product of the number of iterations, the number of machines and the small dimension. It is therefore comparable to SGD which requires, at each iteration, the number of machine times the small dimension communications. The difference is that we have at most n iterations (the smallest dimension) to get an accurate solution, whereas SGD requires approximately $1/\epsilon$ iterations (for a tolerance ϵ).

The computation time is composed of three main parts. The first is the update of the solution of the linear system on the driver (this step has the exact same complexity than the one for the sequential algorithm and does not depend on B). The second is a term of communication between machines due to the broadcast of the residual (when B is 1 it disappears). The last one comes from the computation of the correlation between features, and is inversely proportional to the number of machines. In the case where B is p, computing the correlations is not a problem anymore.

Finally we can analyze the complexity of this algorithm in a PRAM model, where the communication cost is absent. The depth would be the computation time with $B = p$ processors, so it has logarithmic depth since p is the dimension that matters ($p \gg n$). In addition, there is asymptotically the same amount of work as in the sequential implementation.

5 D-LARS for 'Tall and Skinny' matrices ($p \ll n$)

5.1 Set up

For tall and skinny matrices, symmetrically with short and fat, we assume that $p \ll n$, that p^2 fits in memory but n doesn't. Therefore, we must store the data by rows with each of the B machines containing m rows – We assume mp fits in memory ($m \times B = n$). Once again, we want our algorithm to

- Scale on n, the large dimension
- Avoid All-to-All communications

- Not broadcast more than a constant number of vector of size n at each iteration to every machine

Because p^2 fits in memory and $I \leq p$, we will also store the covariance matrix on the driver. To be able to easily solve the linear system giving the direction $\delta_k \leftarrow (X_{A_k}^T X_{A_k})^{-1} X_{A_k}^T r_k$, we store the Cholesky decomposition of the covariance matrix on the driver and update it at each iteration. We will also store the current correlation with the residual and the coefficients of previous iterations on the driver. Each machine will only store its m rows.

The main difference is that in this new framework, computing one correlation already requires communication between machines. Therefore, we will prefer doing all the correlations at the same time to take more advantage of the distributed framework.

5.2 Algorithm

Algorithm 7 Tall and Skinny - Distributed Least Angle Regression

procedure D-LARS(X, y) ▷ X n x p matrix, y vector of size n

  Normalize(X) ▷ Map+AllReduce

  Compute and broadcast $\bar{y}$ ▷ Map + Reduce
  for i in 1 to n **do** ▷ Map
    $r_1^{(i)} \leftarrow y^{(i)} - \bar{y}$
    Emit $(X^{(i)T} r_1^{(i)})$
  end for
  $X^T r_1 \leftarrow \sum_i X^{(i)T} r_1^{(i)}$ ▷ p Reduce
  $j_1, cor_1, s_1 \leftarrow \max(j, |X_j^T r_1|, 2 * 1_{X_j^T r_1 > 0} - 1)$ ▷ Locally

  Init(j_1, X_{j_1})

  for k = 1 to n-1 **do**

    $\delta_k \leftarrow$ **ComputeDelta**(cor_k, X_{A_k}, s) ▷ Locally
    Broadcast δ_k
    Compute $X^{(i)T} r_k^{(i)}$ and $X^{(i)T} X_{A_k}^{(i)} \delta_k$ on every machine ▷ Map
    Sum partial computations to get $X^T r_k$ and $X^T X_{A_k} \delta_k$ ▷ p Reduce
    $j_{k+1}, \alpha_{k+1}, s_{j_{k+1}} \leftarrow$ **ComputeAlpha**$(cor_k, X^T r_k, X^T X_{A_k} \delta_k)$
    Broadcast $\alpha_{j_{k+1}}$

    $A_{k+1} \leftarrow A_k \cup \{j_{k+1}\}$
    $cor_{k+1} \leftarrow (1 - \alpha_{k+1}) cor_k$
    UpdateEstimates$(j_{k+1}, \alpha_{k+1}, X_{A_k}, X_{k+1})$

  end for
return β
end procedure

Init and **ComputeDelta** are implemented as for Short and Fat matrices. **ComputeAlpha** is done locally on the driver with the algorithm below. **UpdateEstimates** is also slightly different and described below.

Algorithm 8 Tall and Skinny ComputeAlpha

procedure ComputeAlpha($cor_k, X^T r_k, X^T X_{A_k} \delta_k$)
  Let $\tilde{X} \leftarrow (X^T X_{A_k})\delta_k$
  for $j \in \bar{A}_k$ **do**
    $\alpha_j^- \leftarrow \left(cor_k - (X^T r_k)_j\right) \left(cor_k - \tilde{X}_j\right)^{-1}$
    $\alpha_j^+ \leftarrow \left(cor_k + (X^T r_k)_j\right) \left(cor_k + \tilde{X}_j\right)^{-1}$
    if $\min(\alpha_j^+, \alpha_j^-) < 0$ **then**
      $\alpha_j \leftarrow \max(\alpha_j^+, \alpha_j^-)$
    else
      $\alpha_j \leftarrow \min(\alpha_j^+, \alpha_j^-)$
    end if
    if $\alpha_j = \alpha_j^+$ **then**
      $\tilde{s}_j \leftarrow +1$
    else
      $\tilde{s}_j \leftarrow -1$
    end if
  end for
  return $\min_j(j, \alpha_j, \tilde{s}_j)$
end procedure

Algorithm 9 Tall and Skinny UpdateEstimates

procedure UpdateEstimates($j_{k+1}, \alpha_{k+1}, X_{A_k}, X_{j_{k+1}}$)
// Update Cholesky factor
  for $i \in [1, n]$ **do**
    Emit $(X_{A_k}^{(i)T} X_{j_{k+1}}^{(i)})$ ▷ Map
  end for
  Sum to compute the new column of the covariance matrix:
  $Cov_{1:k,k+1} \leftarrow X_{A_k}^T X_{j_{k+1}}$ ▷ k Reduce
  $G_{k+1} \leftarrow$ Update Cholesky factor G_k ▷ Locally

// Update residual
  $r_{k+1}^{(i)} \leftarrow r_k^{(i)} - \alpha_{k+1} X_{A_k}^{(i)} \delta_k$ ▷ Map

// Update coefficients
  for $j \in A_k$ **do**
    $\beta_{k+1,j} \leftarrow \beta_{k,j} + \alpha_{k+1}\, \delta_{k,j}$ ▷ Locally
  end for
end procedure

5.3 Complexity

Assuming we have B machine, each having m rows ($mB = n$).

- To normalize the data we have to compute the sum of squares for each column, sum it across the machines and finally broadcast the normalization factors. So this takes $T = O(pm + p\log(B))$ time and $C = O(pB)$ communications. We can also compute $\bar{y}$ with the same method and so it does not change the asymptotic complexity.
- Broadcasting $\bar{y}$ takes time $T = O(\log(B))$ and communication $C = O(B)$.
- Like inside the loop, computing all the correlations using p reduces takes $T = O(mp + p\log(B))$and $C = O(pB)$.
- **Init** takes $T = O(1)$
- For the k^{th} loop:
  - **ComputeDelta** is done locally and takes time $T = O(k^2)$ since the linear system is solved using the Cholesky decomposition of the covariance matrix (as detailed in section 3.4).
  - Broadcasting δ_k takes communication $C = O(kB)$ and time $T = O(k\log(B))$.
  - We compute $X^T r_k$ at each step through p dot products between vectors of size m, before summing across the machines. So this takes $T = O\left(p(m + \log(B)\right)$ time and $C = O(pB)$ communications.
  - The local computation of $X^T X_{A_k}\delta_k$ takes $pm + mk$ time, and the result is a vector of size p as above. So we can compute $X^T X_{A_k}\delta_k$ in $T = O(pm + p\log(B))$ time (since $k \leq p$) and $C = O(pB)$ communications.
  - We compute α_{k+1} locally in $O(p)$ time. It is then broadcast, which takes communication $C = O(B)$ and time $T = O(\log(B))$.
  - **UpdateCholesky** needs to compute the k dot products $X_{A_k}^T X_{j_{k+1}}$ which takes time $T = O\left(k(m + \log(B))\right)$ and $C = O(kB)$ communications. The update of the Cholesky decomposition takes $T = O(k^2)$.

For I iterations, he total complexity is therefore (O omitted)

$$\begin{cases} T &= Ip\left(m + \log(B)\right) + I^3 \\ C &= IpB \end{cases}$$

5.4 Discussion

Let's first look at the complexity in different cases:

- In the case where we want the full path, we have $I = p$ and then:

$$\begin{cases} T = O\left(\frac{np^2}{B} + p^2\log(B) + p^3\right) \\ C = O(p^2 B) \end{cases}$$

- With $B = n$ (a machine for each row) we get:

$$\begin{cases} T = O(p\log(n)I + I^3) \\ C = O(npI) \end{cases}$$

- With $B = p$ and $I = n$:

$$\begin{cases} T = O\left(p^2\log(n) + p^3\right) \\ C = O(p^2 n) \end{cases}$$

We can see that we have the same kind of complexities we got for Short and Fat matrices. All comments regarding Short and Fat then apply (Comparable communication cost to SGD, trade off between communication and decreasing the computation time of the correlations by changing B).

Even though we could have thought that distributed LARS would be more suitable to data matrices stored by column ($n \ll p$), we actually get similar complexities when it is stored by row ($p \ll n$). So those symmetric results suggest that the whole method can also scale very well on n. This is even more surprising given that computations within machines are done very differently.

This very strong result is largely due to the LARS method, that only rely on computing the covariance matrix (correlations). The latter cannot exceed the smallest dimension which guarantees both a small number of iterations and small matrices, as long as one dimension is much smaller than the other.

6 D-LARS for 'Almost Square' matrices ($n \sim p$)

6.1 Set up

For almost square matrices, we assume that both n and p fit in memory but that their square n^2 and p^2 do not. Therefore the data can be stored either by column or row, with each of the B machines containing m columns/rows – We assume mn fits in memory.

In this scenario the difficulty is that after $\sqrt{n}$ iterations, the covariance matrix of the active variables do not fit in memory anymore. Therefore we cannot store the Cholesky factor on the driver if we want more than $\sqrt{n}$ non-zero coefficients.

Thus we split the analysis of the 'almost square' case into two parts, depending on the number of iterations targeted.

First $\sqrt{n}$ iterations The sparsity of the Lasso solution is its main strength, and can also be an objective by itself. Therefore when p is large, it can be a reasonable approach to consider only the first $\sqrt{n}$ steps of the coefficient path (*i.e.* all solutions with at most $\sqrt{n}$ non-zero coefficients). The last iterations will in most cases be useless as we won't get interpretations and the optimal λ is generally implying few nonzero coefficients when p is large.

In that case, the covariance matrix / Cholesky factor fit in memory, hence we can store it on the driver. So we can actually use the algorithms proposed for Short and Fat or Tall and Skinny matrices, which interestingly gives flexibility in the way data is stored (by column or row).

6.2 Computing the whole path

6.2.1 Distributed Cholesky

In this section we describe how we can adapt the algorithm used previously after $\sqrt{n}$ iterations, *i.e.* when the Cholesky factor do not fit in memory anymore.

Updating the Cholesky decomposition When $n \sim p$ we cannot store the covariance matrix on the driver, and it must stay distributed instead. Recall that we only need the Cholesky decomposition of the covariance matrix, and can directly update it at each iteration.

We store the Cholesky factor G (s.t. $C = GG^T$) by row (independently of the rest of the data). At a new step, we get the new column C_{k+1} and we need to compute the last row of G: $g_{k+1,j}$ are computed one by one (and by increasing j) through the formula

$$C_{k+1,j} = \sum_{i \leq j} g_{k+1,i} g_{j,i}$$

Notice that each step is done on a single (but different) machine. So this requires doing $2k$ one-to-one communications to send $C_{k+1,j}$ and the current estimate of g_{k+1}.

Actually if we use combiners and partition the Cholesky factor so that rows with close index are stored on the same machine, then the communication cost is only $C = kB$ (instead of the naive k^2). As before this is done in time $T = k^2$.

Solving the linear system We then need to solve $GG^T x = y$, which is done by first solving $Gz = y$ and then $G^T x = z$.

Notice that the first solve requires exactly the same amount of computations as updating the last row of G described above. The second steps is more costly since we need to broadcast x_j to every machine as soon as its value is computed, which is then done by a reduce. Therefore we solve the linear system in $C = k^2 B$ communications and $T = k(m_G + \log(B))$ (where $m_G < k$ is the maximum number of rows of G stored on the same machine).

Thus the whole step Cholesky update + linear solve takes $C = k^2 B$ communications and $T = k^2 + k \log(B)$ time.

Algorithm The algorithm is based on D-LARS for Short and Fat matrices ($n \ll p$), described in 2. We modify **ComputeDelta** and the Cholesky update in **UpdateEstimates** as described above. We also modify **Init** and **EmitAlpha** so that the correlations between a given variable and the active ones are stored on its machine, thus reducing the communication cost. In addition, note that in this case β is also distributed. The pseudo-code for these functions is given below.

Algorithm 10 Square Init

procedure INIT(j_1, X_{j_1})
    $A_1 \leftarrow \{j_1\}$

    Broadcast X_{j_1}
    for j in 1 to p **do**
        ActiveCor$_j \leftarrow [X_j^T X_{j_1}]$ ▷ Map
        $\beta^{(j)} \leftarrow$ sparse vector of zeros
    end for

end procedure

Algorithm 11 Square EmitAlpha

procedure EMITALPHA($\delta_k, cor_k, \bar{A}_k, X_{A_k}$)
    Broadcast δ_k to every machine

    for $j \in \bar{A}_k$ **do**
        $\alpha_j^- \leftarrow \left(cor_k - X_j^T r_k\right) \left(cor_k - \texttt{ActiveCor}_j \delta_k\right)^{-1}$
        $\alpha_j^+ \leftarrow \left(cor_k + X_j^T r_k\right) \left(cor_k + \texttt{ActiveCor}_j \delta_k\right)^{-1}$
        if $\min(\alpha_j^+, \alpha_j^-) < 0$ **then**
            $\alpha_j \leftarrow \max(\alpha_j^+, \alpha_j^-)$
        else
            $\alpha_j \leftarrow \min(\alpha_j^+, \alpha_j^-)$
        end if
        if $\alpha_j = \alpha_j^+$ **then**
            $\tilde{s}_j \leftarrow +1$
        else
            $\tilde{s}_j \leftarrow -1$
        end if
        Emit $(j, \alpha_j, \tilde{s}_j)$ ▷ Map
    end for

end procedure

Algorithm 12 Square UpdateEstimates

procedure UPDATEESTIMATES($j_{k+1}, \alpha_{k+1}, X_{A_k}, X_{j_{k+1}}$)
    Broadcast α_{k+1}

// Update the distributed Cholesky factor
    $G_{k+1} \leftarrow$ Update Cholesky factor G_k ▷ Locally

// Update residual
    $r_{k+1} \leftarrow r_k - \alpha_{k+1} X_{A_k} \delta_k$ ▷ Map+Reduce

// Update coefficients
    for $j \in A_k$ **do**
        $\beta_{k+1}^{(j)} \leftarrow \beta_k^{(j)} + \alpha_{k+1} \delta_{k,j}$ ▷ Map
    end for
end procedure

6.2.2 Solving the linear system by gradient descent

After $\sqrt{n}$ iterations, we cannot store the covariance matrix on the driver and solving the linear system, even with the use of distributed Cholesky decomposition, ends up with a lot of communications.

To decrease the communication cost as well as the computation time, we propose the use of approximate solutions for the system. This would be done after the first $\sqrt{n}$ iterations, and we suppose that after this point, using approximate coefficient directions would not much the path. We can see our linear system as a quadratic optimization problem [8], by noticing that solving the following system in δ:

$$X^T X \delta = X^T r$$

is the same as finding the minimum of f:

$$f(\delta) = \frac{1}{2} \delta^T X^T X \delta - \delta^T X^T r + c$$

because its gradient is:

$$\nabla f(\delta) = X^T X \delta - X^T r$$

We can then use any optimization technique to solve this problem, which would provide our next estimate of δ_k. By using Gradient Descent or SGD with the last estimate as a warm start, we can get a good approximation of the solution without large communication costs, and still guaranteeing sparsity in the coefficients (β).

By storing the j^{th} row of $X^T X$ on the same machine as the j^{th} column x_j, we can easily perform the updates. We need to broadcast the residual and δ to every machine containing active variables, and send the computed δ back. The other steps involve cheap computation time and communication cost. With standard gradient descent, each step requires more computations but takes more advantage of the parallelization to compute the update. Whereas SGD does not use parallelization but could converge with less than k steps.

6.2.3 Incremental Forward Stagewise approximation

Incremental Forward Stagewise Regression (FS_ϵ) [5] is another path-based linear regression algorithm that provides results very similar to Lasso and LARS. Although it was inspired by boosting procedures, the general idea is very similar to LARS, but with a progression by small *stages* along the path. More precisely, it repeats the following steps, with $\epsilon > 0$ a small constant:

- Find the predictor X_j most correlated with the residual r_k
- Update its coefficient $\beta_j \leftarrow \beta_j + \epsilon\, s_{jk}$, where s_{jk} is the sign of the correlation between X_j and the residual
- Update the residual $r_{k+1} \leftarrow r_k - \epsilon\, s_{jk} X_j$

Therefore this method is very similar to LARS but does not need to compute the coefficient directions (*i.e.* to solve a linear system). Hence we propose to pursue our algorithm, after the first $\sqrt{n}$ iterations, with this discretized version.

A naive implementation would broadcast X_j or the new residual r_{k+1} at each step, but this could drastically increase the communication cost since we cannot bound the number of iterations. However, we could use some heuristics to avoid that, or increase the stage size ϵ. This would result in reducing the number of iterations (speed up), but we would only get a discrete path. In addition, we could try other approximations inspired by stochastic gradient descent. For example if the data was stored by row, we could perform many steps on each machine, before merging the results.

Conclusion

We have shown that using specific methods to distribute a targeted problem can lead to better parallelization of the algorithm. For the Lasso, using the piece-wise linear property of its coefficient path, we have been able to outperform generic gradient based solving methods, while satisfying useful objectives. Adapting Least Angle regression has resulted in various advantages: it guarantees sparsity and computes the full coefficient path with almost no overhead.

We have shown ways to parallelize it for different types of distributed matrices. When one dimension is largely smaller than the other, it is very well adapted since the number of iterations is at most that small dimension, and the covariance matrix can be stored on the driver. For almost square matrices, the computation of the full path is more costly but some approximations could speed up the algorithm. Further studies should focus on this harder case.

This analysis gives new perspective to tackle other L1 penalized versions of machine learning algorithms, such as logistic regression or nearest neighbors [5]. This approach should also motivate the design of other path seekers that approximate famous machine learning algorithms in distributed framework, including the partial least square for Ridge regression or the Generalized Path Seeking (GPS) [4], which can approximate the Lasso path for any convex loss criterion and is also used to solve Gradient Boosting. The latter can solve more general problems with a wide range of penalization (elastic net, ridge), but, like forward stagewise, it does not take advantage of the piece-wise linearity of the Lasso path.

References

[1] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

[2] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[3] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[4] Jerome H Friedman. Fast sparse regression and classification. *International Journal of Forecasting*, 28(3):722–738, 2012.

[5] Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.

[6] M. Lichman. UCI machine learning repository, 2013.

[7] Shai Shalev-Shwartz and Ambuj Tewari. Stochastic methods for l 1-regularized loss minimization. *The Journal of Machine Learning Research*, 12:1865–1892, 2011.

[8] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[9] Jichuan Zeng, Haiqin Yang, Irwin King, and Michael R Lyu. A comparison of lasso-type algorithms on distributed parallel machine learning platforms.

Appendix: Code sample

The full Databricks notebook is available at `http://bit.do/dlars_databricks`.

```
// INIT

// Normalize data ------------------------------------------------------------
val data = data0.map{case(idx, col) =>
  val m = col.sum / col.length
  (idx, col.map{v => v - m})
}.map{case(idx, col) =>
  val std = Math.sqrt(col.map{v => v*v}.sum)
  (idx, col.map{v => v / std})
}.cache()

// Useful variables ----------------------------------------------------------
// nb of features
val p = data.count().toInt
// nb of observations
val N_obs = labels.length
// definition of 'zero'
val epsilon = 1e-10
// control prints
val verbose = true

// Init residuel -------------------------------------------------------------
val mean_labels = labels.sum / labels.length
var res = labels.map{v => v - mean_labels}
//var vec_res = Vectors.dense(res)
var coefs = List(Array.fill(p)(0.0))

// Find first variable to add, which maximizes correlation with res ------------
val max_corr = data.map{case(idx, v) =>
  val corr = mydot(v, res)
  (idx, Math.abs(corr), corr)
}.takeOrdered(1)(Ordering[Double].reverse.on(_._2))(0)
// idx of the next variable to add
var new_feat = max_corr._1
// current correlation btw active set and residuel
var cur_corr = max_corr._2
// indices of the variables in the active set
var active = TreeSet(new_feat)
// init variable for later, suppose to hold (idx, alpha)
var min_alpha = (0, 0.0, 0.0)

var corr_signs = Array.fill(p)(0.0)
corr_signs(new_feat) = bool2dbl(max_corr._3 > 0)
var indexer = Map((new_feat, 0))

// init covariance matrix, its diagonal has only ones
var cov_matrix = breeze.linalg.DenseMatrix.eye[Double](1)
```

```
print("First variable: ")
println(new_feat)
```

```
// LOOP

var k = 0
while(k < Math.min(p, N_obs) && cur_corr > epsilon){
  k += 1
  System.out.println("Step " + k )

  val unactive_cols = data.filter(line => !(active contains line._1))
  val active_cols = data.filter(line => (active contains line._1))

  // Compute new direction -------------------------------------------------------
  val dir = (cov_matrix \ create_cor_vector(active, cur_corr, corr_signs,
      indexer)).toArray

  if(verbose) { System.out.println("dir :" + dir.mkString("<", ",", ">")) }

  if(k < Math.min(p, N_obs)){ // need to compute alpha
    // Compute directed feature ---------------------------------------------------
    // this is X_Ak . delta_k
    val dir_feat = active_cols.map{
        case(idx, v) => scal_mult(v, dir(indexer(idx)))
    }.reduce{
        // add vectors element by element
        case(v1, v2) => v_add(v1, v2, 1.0)
    }

    // Compute alpha --------------------------------------------------------------
    val all_alpha = unactive_cols.map{ case(i, v) =>
      val res_cor = mydot(v, res)
      val dir_cor = mydot(v, dir_feat)
      val a1 = (cur_corr - res_cor)/(cur_corr - dir_cor)
      val a2 = (cur_corr + res_cor)/(cur_corr + dir_cor)
      if(a1 > 0 && a2 > 0){
        (i, Math.min(a1, a2), bool2dbl(a1 < a2))
      } else {
        (i, Math.max(a1, a2), bool2dbl(a1 > a2))
      }
    }
    min_alpha = all_alpha.filter{x => x._2 >
        0}.takeOrdered(1)(Ordering[Double].on(_._2))(0)
    System.out.println("Adding variable " + min_alpha._1)

    // check if res is in Span(A_k), if so, alpha = 1
    if(min_alpha._2 > 1) {
      min_alpha = min_alpha.copy(_2 = 1)
    }

    // update residual
```

```
    res = v_add(res, dir_feat, - min_alpha._2)
  }
  else {
    // this is the last iteration
    min_alpha = (0, 1.0, 0.0) // min_alpha._1 does not exist for the last iteration
  }

  // update coefs values
  val cur_coefs = Array.fill(p)(0.0)
  for(i <- active){
    // coefs(0) is the last value of the coef vector
    cur_coefs(i) = coefs(0)(i) + min_alpha._2 * dir(indexer(i))
  }

  // save coefs
  coefs = cur_coefs :: coefs
  if(verbose){ System.out.println(cur_coefs.mkString("<", ",", ">")) }

  if(verbose) { System.out.println("Alpha step: " + min_alpha._2 + "\n") }

  if(k < Math.min(p, N_obs)) {
    // update for next step
    new_feat = min_alpha._1
    cur_corr = (1 - min_alpha._2 ) * cur_corr
    corr_signs(new_feat) = min_alpha._3

    // Update covariance matrix ------------------------------------------------
    val new_feat_val = data.filter(_._1 == new_feat).collect()(0)
    val new_entries = active_cols.map{
      case(idx, v) => (indexer(idx), mydot(v, new_feat_val._2))
    }.collect().sortWith(_._1 < _._1).map{case(i,v) => v}

    val vec1 = new breeze.linalg.DenseMatrix(new_entries.length, 1, new_entries)
    val vec2 = new breeze.linalg.DenseMatrix(1, new_entries.length+1, (new_entries
        :+ 1.0))
    cov_matrix = breeze.linalg.DenseMatrix.horzcat(cov_matrix, vec1)
    cov_matrix = breeze.linalg.DenseMatrix.vertcat(cov_matrix, vec2)

    active = active + new_feat
    indexer += (new_feat -> k)
  }
}
```