LARS Library: Least Angle Regression Stagewise Library

Frank Vanden Berghen
IRIDIA, Université Libre de Bruxelles
fvandenb@iridia.ulb.ac.be

November 22, 2005

Let's assume that we want to perform a simple identification task. Let's assume that you have a set of $n+1$ measures $(x_{(1)}, x_{(2)}, \ldots, x_{(n)}, y)$. We want to find a function (or a *model*) that predicts the measure y in function of the measures $x_{(j)},\ j = 1, \ldots, n$. The vector $x \in \Re^n = (x_{(1)}, \ldots, x_{(n)})^t$ has several names:

- x is the *regressor vector*.
- x is the *input*.
- x is the vector of independent variables.

y has several names:

- y is the *target*.
- y is the *output*.
- (y is the dependant variable).

The pair (x, y) has several names:

- (x, y) is an *input*$\leftrightarrow$*output* pair.
- (x, y) is a *sample*.

We want to find f such that $y = f(x)$. Let's also assume that you know that $f(x)$ belongs to the family of linear models. I.e. $f(x) = \sum_{j=1}^{n} x_{(j)}\beta_j + p =< x, \beta > +p$ where $< \cdot, \cdot >$ is the dot product of two vectors and p is the *constant term*. You can also write $f(x) = x^t\beta + p$ where $\beta \in \Re^n$ and p are describing $f(x)$. We will now assume, without loss of generality, that $p = 0$. We will see later how to compte p if this is not the case. β is the *model* that we want to find. Let's now assume that we have many *input*$\leftrightarrow$*output* pairs. i.e. we have

$$\begin{aligned} (x_{(1)}, \ldots, x_{(n)})_{(1)} = X_{(1)} &\leftrightarrow y_1 \\ X_{(2)} &\leftrightarrow y_2 \\ \vdots &\quad \vdots \\ X_{(m)} &\leftrightarrow y_m \end{aligned} \tag{1}$$

We want to compute β such that $\forall i\ \ X_{(i)}^t\beta = y_i$. Using matrix notation, β is the solution of:

$$X\beta = y \tag{2}$$

where $X \in \Re^{m \times n}$ is a matrix containing on each line a different *regressor* and $y \in \Re^m$ contains all the targets. The columns of X contain the independent variables or, in short, the *variables*. If $m \leq n$, we can compute β using a simple Gauss-Jordan elimination. That's not very interesting. We are usually more interested in the case where $m >> n$: when the linear system $X\beta = y$ is over-determined. We want to find β^* such that the Total Prediction Error (*TPError*) is minimum:

$$TPError(\beta^*) = \min_{\beta} (TPError(\beta)) \tag{3}$$

If we define the total prediction error *TPError* as

$$\textit{Sum of Squared Errors}(\beta) = \sum_{i=1}^{m} (X_{(i)}\beta - y_i)^2 = (X\beta - y)^2 = \|X\beta - y\|_2$$

where $\| \cdot \|_2$ is the $L2$-norm of a vector, we obtain the *Classical Linear Regression* or *Least Min-Square*(LS):

$$\textit{Classical Linear Regression: } \beta^* \text{ is the solution to: } \min_{\beta} (\|X\beta - y\|_2) = \min_{\beta} \left((X\beta - y)^2\right)$$

Such a definition of the total prediction error *TPError* can give un-deserved weight to a small number of bad *samples*. It means that **one** sample with a large prediction error is enough to change completely β^* (this is due to the squaring effect that "amplifies" the contribution of the "outliers"). This is why some researcher use a $L1 - norm$ instead of a $L2 - norm$:

$$\textit{Robust Regression: } \beta^* \text{ is the solution to: } \min_{\beta} (\|X\beta - y\|_1) = \min_{\beta} \left(\sum_{i=1}^{m} |X_{(i)}\beta - y_i| \right)$$

where $|\cdot|$ is the absolute value. We will not discuss here *Robust regression* algorithms. To find the solution of a *classical Linear Regression*, we have to solve:

$$\begin{aligned} Error^2(\beta^*) &= \min_{\beta} \left\{ (X\beta - y)^2 \right\} \\ &= \min_{\beta} \left\{ (X\beta - y)^t (X\beta - y) \right\} \\ &= \min_{\beta} \left\{ (\beta^t X^t - y^t)(X\beta - y) \right\} \\ &= \min_{\beta} \left\{ (\beta^t X^t X\beta - y^t X\beta - \beta^t X^t y + y^t y) \right\} \end{aligned}$$

Since $y^t X\beta$ is a scalar, we can take its transpose without changing its value: $y^t X\beta = (y^t X\beta)^t = \beta^t (y^t X)^t = \beta^t X^t y$. We obtain:

$$Error^2(\beta^*) = \min_{\beta} \left\{ \beta^t X^t X\beta - 2\beta^t X^t y \right\}$$

$$\begin{aligned} Error^2(\beta^*) \text{ is minimum} \Leftrightarrow & \quad \frac{\partial Error^2(\beta)}{\partial \beta}(\beta^*) = 0 \\ \Leftrightarrow & \quad 2X^t X\beta^* - 2X^t y = 0 \\ \Leftrightarrow & \quad X^t X\beta^* = X^t y \end{aligned} \tag{4}$$

The equation 4 is the well-known *normal equation*. On the left-hand-side of this equation we find $X^tX = C \in \Re^{n\times n}$: the *correlation matrix*. C is symmetric and semi-positive definite (all the eigenvalues of C are ≥ 0). The element $C_{ij} = \sum_{k=1}^{m} X_{ik}X_{jk} =< X_i, X_j >= X_i^tX_j$ is the correlation between column i and column j (where X_i stands for column i of X). The right-hand-side of equation 4 is also interesting: it contains the univariate relation of all the columns of X (the *variables*) with the target y.

We did previously the assumption that the constant term p of the model is zero. p is null when $mean(y) = 0$ and $mean(X_i) = 0$ where $mean(\cdot)$ is the mean operator and X_i is the i^{th} column of X. Thus, in the general case, when $p \neq 0$, the procedure to build a model is:

- Compute $mean(y), std(y), mean(X_i), std(X_i)$ where $std(\cdot)$ is the "standard deviation" operator.
- Normalize the target: $\tilde{y} = \dfrac{y - mean(y)}{std(y)}$
- Normalize the columns of X: $\tilde{X_i} = \dfrac{X_i - mean(X_i)}{std(X_i)}$
- Since the columns of $\tilde{X}$ have zero mean, and since $\tilde{y}$ has zero mean, we have $p = 0$. We can thus use the *normal equations:* $\tilde{X}^t\tilde{X}\tilde{\beta} = \tilde{X}^t\tilde{y}$ to find $\tilde{\beta}$. All the columns inside $\tilde{X}$ have a standard deviation of one. This increase numerical stability.
- Currently, $\tilde{\beta}$ has been computed on the "normalized columns" $\tilde{X_i}$ and $p = 0$. Let's convert back the model so that it can be applied on the un-normalized X and y:
  * $\beta_i = \dfrac{std(y)}{std(X_i)}\,\tilde{\beta}_i, \;\; i = 1, \ldots, n$
  * $p = mean(y) - \sum_{i=1}^{n} mean(X_i)\beta_i$

From now on, in the rest of this paper, we will always assume that $p = 0$ (data has been normalized).

One simple way to solve the *normal equations* 4 is the following:

- Compute a Cholesky factorization L of $C = X^tX$. i.e. Compute L such that L is a lower triangular matrix and such that $L^tL = C$.
- Let's define $v := L\beta$. Then we have: $C\beta = L^tL\beta = L^tv = X^ty$ (the last equality comes from equation 4). This last equation ($L^tv = X^ty$)can be solved easily because L^t is triangular. Once we have found v, we solve $L\beta = v$ to find β (this is also easy because L is triangular).

Unfortunately, the above algorithm is not always working properly. Let's consider the case where $X_i = X_j$. i.e. the variables i and j are 100% linearly correlated together. In this case, we will have $\beta_iX_i + \beta_jX_j = \beta_iX_i + \beta_jX_i = (\beta_i + \beta_j)X_i$. β_i can be any value at the condition that $\beta_j := -\beta_i$ (so that we obtain $\beta_i + \beta_j = 0$ and none of the columns i and j are meaningful to predict y). Thus, β_i can become extremely large and give undeserved weight to the column i. We have one degree of liberty on the β_i's. This difficulty arises when:

- A column of X is a linear combination of other columns.
- The rank of A is lower than n.
- The matrix $C = X^t X$ is not of full rank n.

This problem is named the "multi-colinearity problem". It is illustrated in figure 1 and 2.

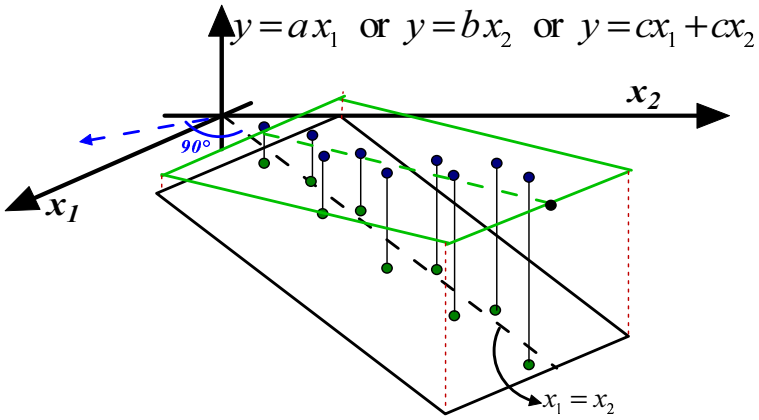

Figure 1: Stable Model

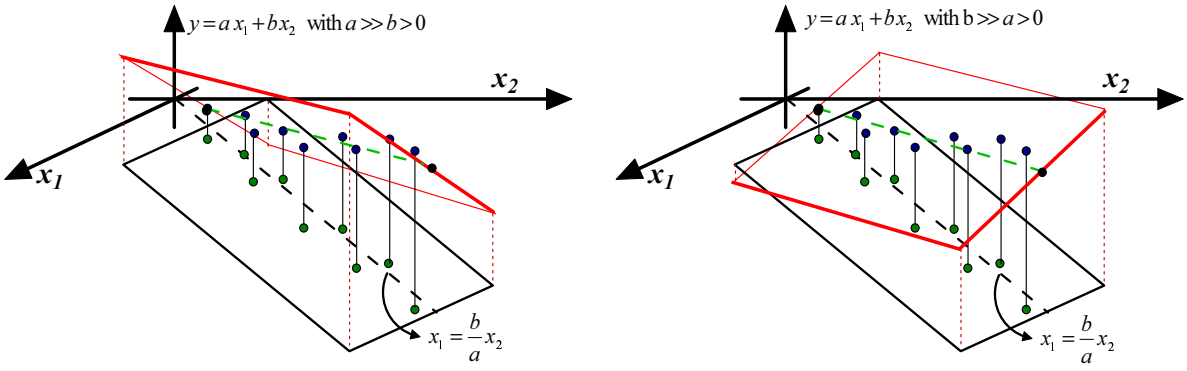

Figure 2: Unstable Models

The blue points inside figures 1 and 2 are the samples. The green points are the lines of $X \in \Re^{9\times 2}$ (the *inputs*) and the heights of the blue points are the targets ($y \in \Re^9$)(the *output*). There is a linear correlation between X_1 and X_2: $X_1 \approx C^{te} X_2$. In the "blue direction" (perpendicular to the black dashed line), we do not have enough reliable information to decide if the output y should increase or decrease. Inside figures 1 and 2, a model β is represented by a plane. In this situation, the ideal model that we want to obtain, is drawn in green in figure 1 (the plane/model is "flat": there is no un-due variation of y in the "blue direction"). The equation of the ideal model is either: $y = ax_1$ or $y = bx_2$ or $y = cx_1 + cx_2$. Note that, inside figure 2 (and also 1), we do NOT have exactly $X_1 = C^{te} X_2$ (the green point are not totally on the dashed line $x_1 = \frac{b}{a} x_2$). Thus, we theoretically have enough information in the "blue direction" to construct a model. However exploiting this information to predict y is not a good idea and leads to unstable models: a small perturbation of the data (X or y) leads to a completely different model: Inside figure

2, we have represented in red two models based on small perturbations of X and y. Usually, unstable models have poor accuracy. The main question now is: How to stabilize a model? Here are two classical solutions:

1. Inside figure 1, the 'information' that is contained in X_1 is duplicated in X_2. We can simply drop one of the two variables. We obtain the models $y = ax_1$ or $y = bx_2$. The main difficulty here is to select carefully the columns that must be dropped. The solution is the "LARS/Lasso Algorithm" described later in this paper.

2. The other choice is to keep inside the model $y = \beta_1 x_1 + \beta_2 x_2$ all the variables ($\beta_1 \neq 0$ and $\beta_2 \neq 0$). However, in this case, we must be sure that:

   - $\beta_1 \approx \beta_2$
   - β_1 and β_2 are small values (I remind you that β_1 and β_2 can become arbitrarily large).

   We must find an algorithm to compute β that will "equilibrate" and "reduce" the β_i's:

   $$\beta^* \text{is the solution of} = \min_{\beta} \left\{ (X\beta - y)^2 + \lambda\beta^2 \right\}$$

   $$\Leftrightarrow (X^t X + \lambda I)\beta^* = X^t Y \tag{5}$$

   This technique is named "Ridge regression". When λ increases, all the β_i are "equilibrated" and "pushed" to zero as illustrated in figure 3. We will discuss later of a way to compute an

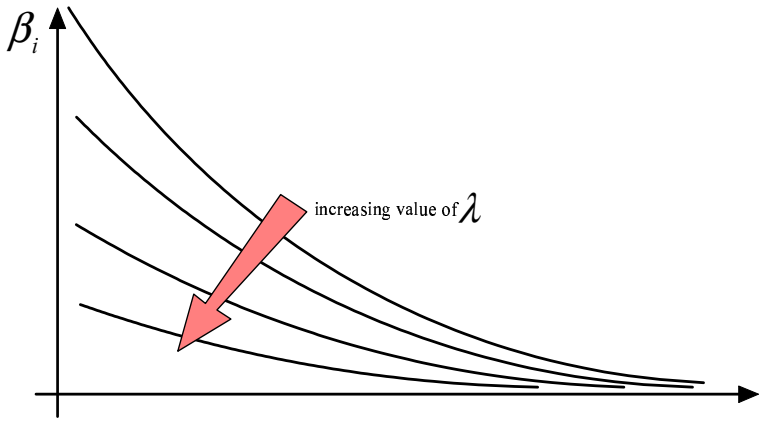

Figure 3: The β_i's sorted from greatest to smallest for different value of λ.

   optimal regularization parameter λ. Inside figure 1, we obtain the model $\beta = (\beta_1 \ \ \beta_2)^t = (c \ \ c)^t$ and we have thus $y = cx_1 + cx_2$ with c "small".

The "Ridge regression" technique is particularly simple and efficient. It has a nice graphical explanation. The "Ridge regression" technique is in fact searching for β^* that minimizes $Q(\beta) =$ Squared Error$(\beta) = (X\beta - y)^2$ under the constraint that $\|\beta^*\|_2 < r$. This definition is illustrated on the left of figure 4. We seek the solution β^* of the minimization problem:

$$\min_{\beta \in \Re^n} Q(\beta) \equiv g_k^t \beta + \frac{1}{2}\beta^t H \beta \quad \text{subject to } \|\beta\|_2 < r \tag{6}$$

(with $H = X^t X$ and $g = -X^t y$) To solve this minimization problem, we first have to rewrite the constraint: $c(\beta) = \frac{1}{2}r^2 - \frac{1}{2}\beta^t\beta < 0$. Now, we introduce a Lagrange multiplier λ and the

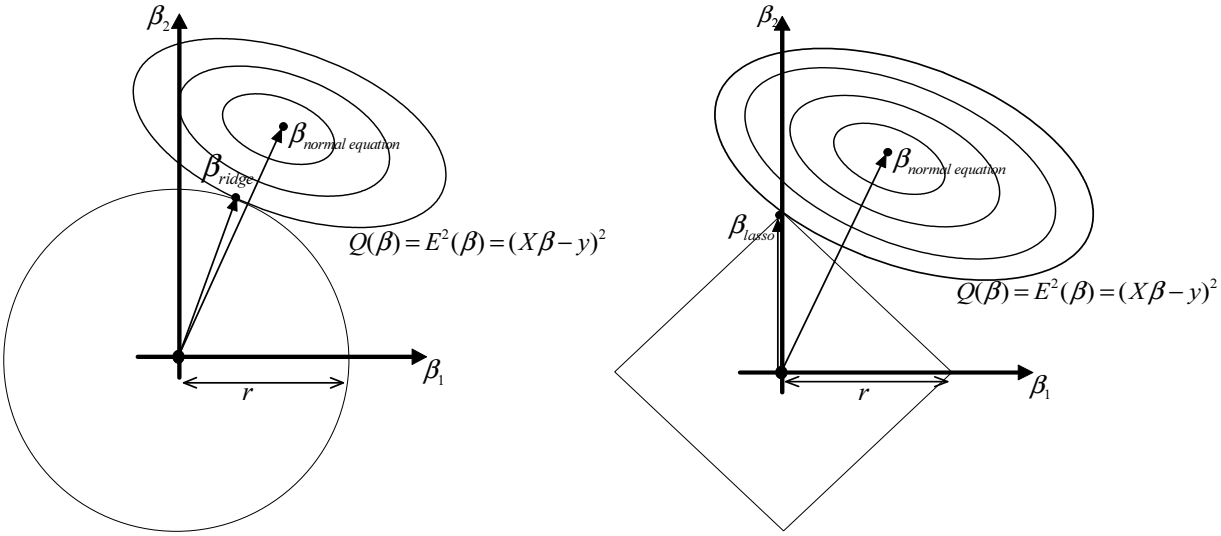

Figure 4: On the left: Illustration of Ridge Regression. On the right: Illustration of Lasso

Lagrange function:

$$\mathcal{L}(\beta, \lambda) = Q(\beta) - \lambda c(\beta)$$

Using the Lagrange First order optimality criterion, we know that the solution (β^*, λ^*) to this optimization problem is given by:

$$\begin{aligned}
& \nabla_\beta \mathcal{L}(\beta^*, \lambda^*) && = 0 \\
\Leftrightarrow \quad & \nabla Q(\beta^*) - \lambda^* \nabla c(\beta^*) && = 0 \\
\Leftrightarrow \quad & H\beta^* + g + \lambda^* \beta^* && = 0 \\
\Leftrightarrow \quad & (H + \lambda^* I)\beta^* + g && = 0 \\
\Leftrightarrow \quad & (X^t X + \lambda^* I)\beta^* - X^t Y && = 0 \\
\Leftrightarrow \quad & (X^t X + \lambda^* I)\beta^* && = X^t Y
\end{aligned}$$

which is equation 5. Thus the constraint $\|\beta\|_2 < r$ prevents the β_i's from becoming arbitrarily large. As illustrated in figure 3, the ridge regression algorithm "pushes" all the β_i's towards zero but none of the β_i actually gets a null value. It's interesting to have $\beta_i = 0$ because, in this case, we can completely remove from the model the column i. A model that is using less variables has many advantages: it is easier to interpret for a human and it's also more stable. What happens if we replace the constraint $\|\beta\|_2 < r$ with the constraint $\|\beta\|_1 = \sum_{i=1}^{n} |\beta_i| < r$? See the right of illustration 4. You can see on the figure that $\beta_{lasso} = (\beta_1^*, \beta_2^*)$ with $\beta_1^* = 0$. You can see on figure 5 that, as you decrease r, more and more β_i^* are going to zero. The Lasso regression is defined by: β^* is the solution of the minimization problem:

$$\min_{\beta \in \Re^n} (X\beta - y)^2 \quad \text{subject to } \|\beta\|_1 < r \tag{7}$$

As you can see, computing different β's (with a different number of null β_i's) involves solving several times equation 7 for different values of r. Equation 7 is a QP (Quadratic Programming) problem and is not trivial to solve at all. The Lasso algorithm, as it is described here, is thus very demanding on computer resources and is nearly of no practical interest. Hopefully, there exist another, more indirect, way to compute at a very little cost all the solutions β of the Lasso Algorithm for all the values of r. A modification of the LARS algorithm computes all the

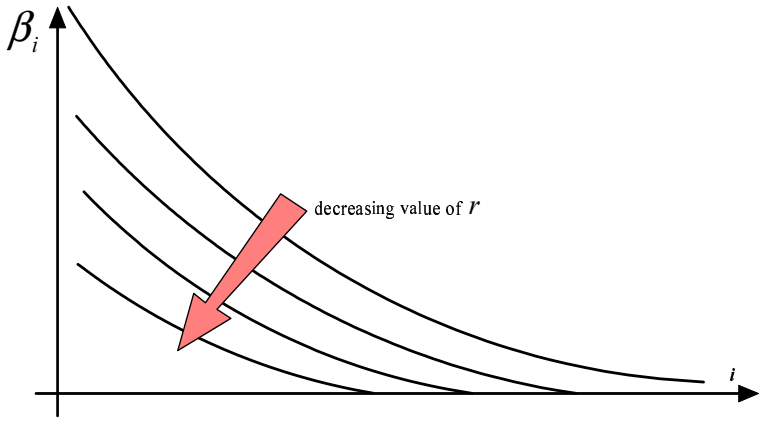

Figure 5: The β_i's sorted from greatest to smallest for different values of r

Lasso solution in approximatively the time needed to compute and solve the simple "normal equations" (equation 4) (the algorithmic *complexity* of the two procedures is the same). LARS stands for "Least Angle Regression laSso". We will not describe here the LARS algorithm but we will give an overview of some of its nicest properties.

The LARS algorithm is a refinement of the FOS (Fast Orthogonal Search) algorithm. We will start by describing the FOS algorithm. The FOS (and the LARS algorithm) is a forward stepwise algorithm. i.e. FOS is an iterative algorithm that, at each iteration, includes inside its model a new variable. The set of variables inside the model at iteration k is the "active set" $\mathcal{A}_k$. The set of variables outside the model is the "inactive set" $\mathcal{I}_k$. $X_{\mathcal{A}}$ is a subset of column of X containing only the active variables at iteration k. $X_{\mathcal{I}}$ is a subset of column of X containing only the inactive variables at iteration k.

Let's first illustrate the equation 2: $X\beta = y$. Equation 2 can be rewritten:

$$X\beta = \beta_1 \begin{pmatrix} \vdots \\ X_1 \\ \vdots \end{pmatrix} + \beta_2 \begin{pmatrix} \vdots \\ X_2 \\ \vdots \end{pmatrix} + \cdots + \beta_n \begin{pmatrix} \vdots \\ X_n \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ y \\ \vdots \end{pmatrix} \tag{8}$$

If $X \in \Re^{m \times n}$, then $X_i \in \Re^m$ is a column of X. X_i can also be seen as a vector in a m dimensional space and $\beta \in \Re^n$ is the best linear combination of the X_i vectors to reach the target y. The "vector view" of equation 8 is illustrated in figure 6 where

$$X \in \Re^{3\times 2} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \\ 0 & 0 \end{pmatrix} \text{ and } \beta = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \text{ and } y = \begin{pmatrix} 3 \\ 2 \\ 3 \end{pmatrix}$$

The FOS algorithm does the following:

1. Let's define the residual prediction error at step k as E_k. We have $E_0 := y$. Normalize the variables X_i and the target y to have $\|X_i\|_2 = 1$ and $\|y\|_2 = 1$. Set $k := 0$, the iteration counter.

2. This step is illustrated in figure 7, Part 1. We will add inside $\mathcal{A}_k$ the variable j that is "the most in the direction of E_k". i.e. the "angle" α_j between X_j and E_k is smaller than

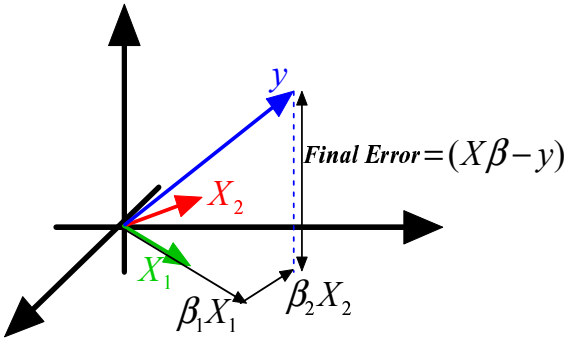

Figure 6: Graphical interpretation of $X\beta = y$

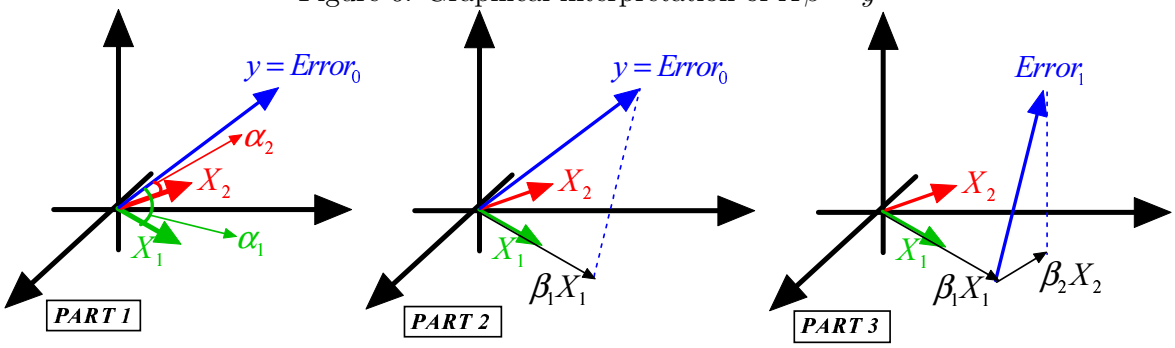

Figure 7: Illustration of the FOS algorithm

all the other angles α_i $(i \neq j)$ between the X_i's and E_k. Using the well-know relation:

$$cos(\alpha_i) = \frac{< X_i, E_k >}{\|X_i\| \, \|E_k\|} \tag{9}$$

Since the variables X_i have been normalized, we obtain:

$$cos(\alpha_i) = C^{te} \; X_i^t E_k \qquad (\text{where } C^{te} = \frac{1}{\|E_k\|}) \tag{10}$$

Note that $X_i^t E_k$ is simply the correlation between variable i and the Residual Error E_k.

Thus the algorithm is: "compute $a \in \Re^n := X_{\mathcal{I}}^t E_k$. Find j such that

$$a_j = \max_i \{a_i\} \tag{11}$$

Add the variable j to the active set $\mathcal{A}_k$. Remove variable j from the inactive set $\mathcal{I}_k$".

3. This step is illustrated in figure 7, Part 2. We now compute β_j, the length of the "step" in the direction X_j (inside the example $j = 1$). Let's project (orthogonally) E_k inside the space that is "spanned" by the active variables. The projection of E_k is named E_{proj}. We obtain $\beta_j = \frac{\|E_{proj}\|}{\|X_j\|} = \|E_{proj}\|$ since X_j is normalized. If the projection operator fails (X_j is inside the space spanned by the active set $\mathcal{A}_{k-1}$ of variables), remove j from the active set $\mathcal{A}_k$ and go to step 2.

4. Deflation: update of the Residual Error: $E_{k+1} := E_k - E_{proj}$.

5. If $\mathcal{I}_k$ is empty then stop, otherwise increment k and go to step 2.

Let's define n_a, the number of active variable at the end of the algorithm. If you decide to stop prematurely (for example, when $E_{k-1} - E_k \approx 0$), you will have $n_a << n$. The FOS algorithm has several advantages:

- It's ultra fast! Furthermore, the computation time is mainly proportional to n_a. If $n_a << n$, then you will go a lot faster than when using the standard normal equations.
- The algorithm will "avoid" naturally the correlated columns. I remind you that a model that is using two strongly correlated columns is likely to be unstable. If column i and j are strongly correlated, they are both "pointing" to the same direction (the angle between them is small: see equation 9 and figure 6) . If you include inside the active set the column X_i, you will obtain, after the deflation, a new error E_{k+1} that will prevent you to choose X_j at a later iteration. One interpretation is the following: "All the information contained inside the direction X_i has been used. We are now searching for other directions to reach y". It is possible that, at the very end of the procedure, the algorithm still tries to use column X_j. In this case, the projection operator used at step 3 will fail and the variable X_j will be definitively dropped. Models produced with the FOS algorithm will thus be usually very stable.
- The selection of the column that enters the active set is based on the "residual error" E_k at step k (with $E_0 = y$). This selection is thus exploiting the information contained inside the target y. The "target information" is not used by other algorithms that are based on SVD of X or QR factorization of X. The FOS algorithm will thus performs a better variable selection than SVD- or QR-based algorithm. Many other algorithms do not use any deflation. Deflation is important because it allows us to search for variables that explains the part of the target that is still un-explained (the residual error E_k). A good variable selection is useful when $n_a << n$.
- The memory consumption is very small. To be able to use the FOS algorithm, you only need to be able to store in memory the target y, one line of X, and a dense triangular matrix of dimension n_a. Other algorithms based on QR factorization of X requires to have the full X matrix inside memory because the memory space used to store X is used during the QR factorization to store temporary, intermediate results needed for the factorization. Algorithms based on SVD of X are even more memory hungry. This is a major drawback for most advanced applications, especially in Econometrics where we often have $X \in \Re^{10^7 \times 10^4}$. There exists some "out of core" QR and SVD factorizations that are able to work even when X does not fit into memory. However "out of core" algorithms are currently extremely slow and unstable.

There is however one major drawback to the FOS algorithm. It is illustrated in figure 8 where we have:

$$X \in \Re^{3\times 3} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \beta = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \text{ and } y = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \tag{12}$$

At the first iteration, the FOS algorithm selects X_3 because $X_3^t\ y = 2$ is greater than $X_1^t\ y = X_2^t\ y = 1$ (and $E_0 = y$). Thereafter, the FOS algorithm computes the new Residual Error $E_1 = (0\ \ 0\ \ -1)^t$ and get stuck. Indeed, if we add to the active set either X_1 or X_2, the L2-norm

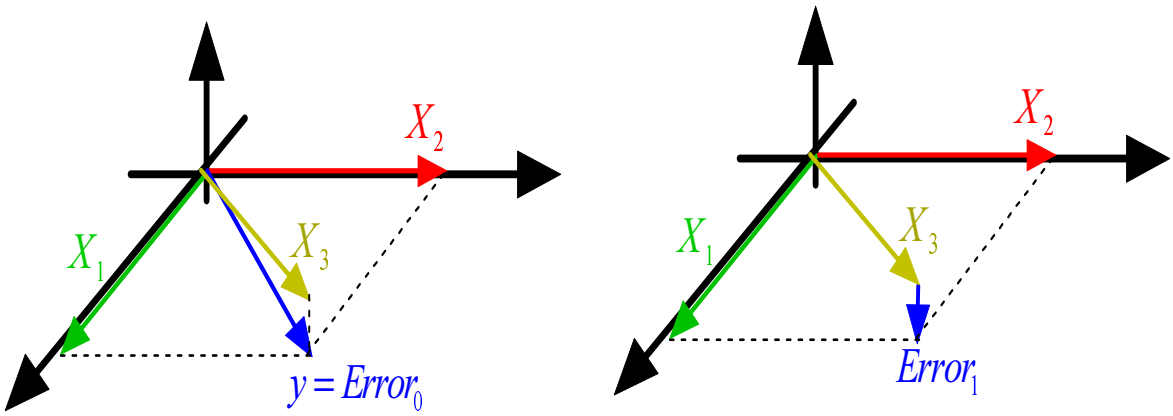

Figure 8: FOS algorithm fails and gives $\beta = (0 \;\; 0 \;\; 1)^t$

of the Residual Error do not decrease: $\|E_2\|_2 = \|E_1\|_2$. There is no way to add a new variable to decrease the Residual Error and thus the algorithm stops. There are mainly two conceptual reasons why the FOS algorithm fails:

1. The FOS algorithm is a forward stepwise algorithm and, as almost all forward stepwise algorithms, it is not able to detect multivariate concepts. In the example illustrated in figure 8, the target is defined precisely to be $Y = X_1 + X_2$. You are able to predict the target accurately if your model is $\beta = (1 \;\; 1 \;\; 0)^t$. The FOS algorithm gives the wrong model $\beta = (0 \;\; 0 \;\; 1)^t$. In this example, the target y is a multivariate concept hidden inside two variables: X_1 and X_2. A forward stepwise algorithm that is able to detect a multivariate concept is very difficult to develop. Inside FOS, the selection of the variable X_j that enters the active set is only based on the univariate relation of X_j with the target Y. On the contrary, Backward stepwise algorithms have no problem to detect multi-variable concepts and should thus be preferred to a Simple,Classical Forward Stepwise Algorithm. However, usually, Backward stepwise algorithms are very time consuming because they need first to compute a full model and thereafter to select carefully which variables they will drop.

   The LARS algorithm with Lasso modification is a forward stepwise algorithm that produces all the solutions of the Lasso algorithm in a computing time proportional to n_a (n_a is the number of active variable at the end of the algorithm). The Lasso algorithm is part of the family of the backward stepwise algorithm (It starts with a full model and remove some variables when r decreases: see equation 17). Thus, the Lasso algorithm is able to discover multi-variable concepts. And thus, the LARS algorithm with Lasso modification is also able to discover multi-variable concepts (since LARS with Lasso gives the same models than the Lasso algorithm). From my knowledge, the LARS algorithm with Lasso modification is the only forward stepwise algorithm able to discover multi-variable concepts. Since, it's a forward stepwise algorithm, it's also very fast.

   On the other side, conjunctive concepts are easy to detect. Let's assume that you have the following:

$$X \in \Re^{3\times 2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \text{ and } \beta = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \text{ and } \textit{constant term}{=}p = -1 \text{ and } y = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

If you create a column $X_3 = X_1 * X_2$ ($X_{3,i} = X_{1,i}X_{2,i}$, $i = 1, \ldots, m$), you obtain:

$$X3 = \begin{pmatrix} 1 \times 0 = 0 \\ 0 \times 1 = 0 \\ 1 \times 1 = 1 \end{pmatrix} \quad \text{and} \quad \beta = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

i.e. You can represent conjunctive concepts with the product of variables. You will thus obtain second order models. Computing second order models usually involves extending the original X matrix with columns that are the product of some of the columns of the original X. This is usually a very memory expansive operation. Indeed if n is the number of column of X, then the extended $X_{extended}$ matrix has $\frac{1}{2}n(n+1)$ columns. The LARS library has been built to allow you to easily create "on the fly" a *regressor* (= one line of $X_{extended}$). i.e. you only need to be able to store in memory one line of $X_{extended}$, not the whole $X_{extended}$ matrix. This allows you to compute at a low memory consumption second, third, fourth,... order models.

2. The FOS algorithm is "too greedy": when the FOS algorithm finds a good direction, it is exploiting this direction "to the maximum" (the length of the steps is $\|E_{proj}\|$). Thereafter there is no "room left" for further improvements, adding other variables. The FOS has been "trapped" in a so-called "local optimum". The LARS algorithm is not so greedy. Its steps lengthes are shorter (see step 3. of the FOS algorithm about "steps"). This means that the different models that are computed at each steps of the LARS algorithm are not 100% efficient. With the same set of active variables, one can compute a better model. The LARS library downloadable on this page allows you to obtain both models:

   - the most efficient one.
   - the one used inside the LARS algorithm to search for the next variables to add inside the active set.

To resume, the LARS algorithm has all the advantages of the FOS algorithm:

1. ultra fast.
2. correlated columns are avoided to obtain stable models.
3. Selection of the active variables based on the target (Deflation).
4. small memory consumption.

Furthermore, the LARS algorithm with Lasso modification is able to discover multivariate concepts easily. For example, the LARS algorithm with LASSO modification finds $\beta = (1 \quad 1 \quad 0)^t$ for the example illustrated in figure 8.

There exists one last question to answer when using the LARS algorithm: When to stop? How to choose n_a? The question is not simple. Several strategies are available inside the LARS library downloadable on this page:

1. Let's define $a(k)$ the maximum correlation of the inactive columns with E_k, the residual error at iteration k (see the definition of a_j in equation 11).

   $$\text{if } ((a(k-10) - a(k)) < s_a \; a(0), \text{ then stop} \tag{13}$$

   where s_a is a user-defined parameter (usually $s_a = .05$) and k is the iteration index.

2.

$$\text{if } ((E_{k-10}^2 - E_k^2) < s_E \, E_0^2, \text{ then stop} \tag{14}$$

where E_k is the residual error at iteration k and s_E is a user-defined parameter (usually $s_E = .01$).

3. Let's define $C_p(k) = E_k^2 - n + 2 * n_a(k)$ where $n_a(k)$ is the number of active variable at iteration k.

$$\text{if } (C_p(k) > max\{C_p(k - s_c), C_p(k - s_c + 1), \ldots, C_p(k - 1)\}, \text{ then stop} \tag{15}$$

where s_c is a user-defined parameter (usually $s_c = 5$) and k is the iteration index.

4. Let's define $minCVError(k)$ as the n-Fold-Cross-Validation-Error of a model $\beta(k)$ that has been regularized using an optimized ridge parameter λ.

$$\text{if } (minCVError(k) > minCVError(k - 1)), \text{ then stop} \tag{16}$$

where k is the iteration index (WARNING: very slow! It involves a full derivative-free optimization algorithm (Brent) at each iteration of the LARS algorithm to find the optimum λ).

My advice is :

1. to choose a very stringent termination test (like the first one with $s_a = .005$) to be sure to have all the informative variables. s_a has a nice and easy interpretation: it's the percentage of the variance inside y that will not be explained.

2. to perform a backward stepwise, using as initial model, the model given by the LARS algorithm. The LARS library has been designed so that such task is very easy to code.

My experience is that stopping criterions based on MDL, Akkaike index or C_p statistics are not reliable.

Methodology

To be able to answer the simple question "How good is my model?" you need a good methodology. Let's assume that we have a set of data X and y. You will cut this data in three parts: a creation set X_{crea}, a validation set X_{val} and a test set X_{test}. The procedure is illustrated in figure 9. The creation set will be used to build a first model. You can build the model β using the normal equation ($X_{crea}^t X_{crea} \beta = X_{crea}^t y_{crea}$) or using the LARS algorithm. Thereafter the model β needs to be stabilized (stabilization is less important if you have built the model with the LARS algorithm but can still improve a little your model). There are mainly two simple ways to stabilize a model: Remove un-informative variables (this is called "backward stepwise") and Ridge Regression.

To remind you, the Ridge Regression is based on equation 5:

$$(X_{crea}^t X_{crea} + \lambda I) \, \beta(\lambda) \; = X_{crea}^t y_{crea} \tag{17}$$

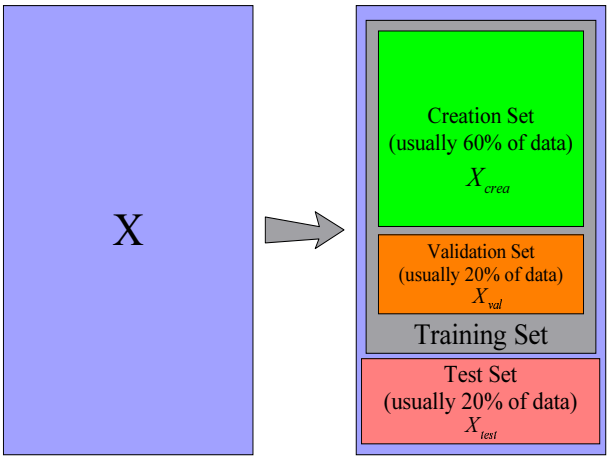

Figure 9: Data are cut into three parts

where λ is the regularization parameter that needs to be adjusted, $\beta(\lambda)$ is the model (depending on the value of λ) and X_{crea} is the creation set. If λ is high, the model will be strongly stabilized. The optimal value of the λ parameter is λ^*. λ^* is the solution to the optimization problem:

$$\min_{\lambda} \left((X_{val} \ \ \beta(\lambda) - y_{val})^2 \right)$$

where $\beta(\lambda)$ is computed from equation 17. We are searching for the value of λ that gives the best model when applied on the validation set. We are improving the "generalization ability" of our model. Since we never used any data from the validation set to build our model, the performance of the model $\beta(\lambda)$ is entirely due to its "generalization ability".

To stabilize your model, you can also completely remove some variables using a "backward stepwise" algorithm. Here is a simple "backward stepwise" algorithm:

1. Let's define $X_{crea}^{(-i)}$, the creation set without the column i. Let's define $sqE_{ref} = (X_{val}\beta - y_{val})^2$ where β is the solution to the standard normal equation: $X_{crea}^t X_{crea} \ \beta \ = X_{crea}^t y_{crea}$. sqE_{ref} is the performance on the validation set of a model β using all the columns of the creation set X_{crea}. Set $i := 0$.

2. We will try to remove the variable i. Compute $\beta^{(-i)}$, the solution to $X_{crea}^{(-i)t} X_{crea}^{(-i)} \ \beta^{(-i)} = X_{crea}^{(-i)t} y_{crea}$. Compute $sqE^{(-i)} = (X_{val}\beta^{(-i)} - y_{val})^2$: the performance of the model $\beta^{(-i)}$ on the validation set X_{val} when the model is built without taking into account the i^{th} column of the creation set X_{crea}. If $sqE^{(-i)} \approx sqE_{ref}$, then the column i did not contain any information and can be dropped: Set $X_{crea} := X_{crea}^{(-i)}$.

3. Choose another i and go back to step 2.

Let's define $\mathcal{A}$, the optimal set of column after the backward stepwise.

Once we have found an optimal ridge parameter λ^* and/or the optimal set $\mathcal{A}$ of informative columns (backward stepwise), we can compute the final model β^* using:

$$(X_{train}^t X_{train} + \lambda^* I) \ \beta^* \ = X_{train}^t y_{train} \tag{18}$$

where X_{train} contains only the columns in $\mathcal{A}$.

WARNING: If you try to use both regularization techniques at the same time, they will "enter in competition". You should then be very careful.

To get an idea of the quality of your final model β^*, you can apply it on the test set:

$$\text{Squared Error on test set}=(X_{test}\ \beta^* - y_{test})^2$$

It's crucial to never use the data in the Test Set X_{test} when building the model because otherwise the results will always be too optimistic. Indeed, if we have used a part of the test set to build the model, we are "cheating": we are not truly estimating the "generalization ability" of our model.

The methodology just described here is a correct approach to a modelization problem. However, this approach does not use a very large validation and test set. Since the validation set is small, the λ parameter and the set $\mathcal{A}$ of active columns will be very approximative. Since the test set is small, the estimation of the quality of the model is also very approximative. To overcome these difficulties, we can use a n-fold-cross-validation technique. If we want a more accurate estimation of the quality of the model, we can build 5 test sets instead of one: see figure 10. It's named a 5-fold-cross-validation. It also means that we have 5 training sets and thus 5 different models. The main point here is that none of these 5 models have seen the data that are inside their corresponding test set.

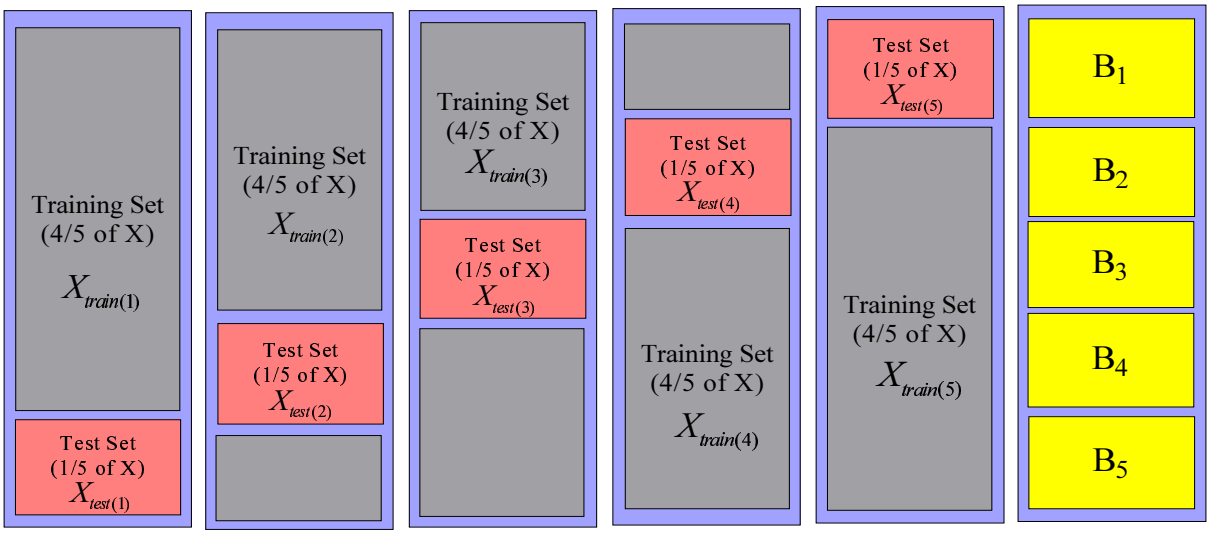

Figure 10: 5 test sets (and 5 training sets))

An estimation of the overall quality of our modelization is:

$$\begin{aligned}\text{overall quality of the modelization}= \\ \text{5-fold-cross-validation error}=\end{aligned} \sum_{i=1}^{5}(X_{test(i)}\beta_{(i)} - Y_{test(i)})^2$$

$$\text{where } \beta_{(i)} \text{ is the solution to } X^t_{train(i)}X_{train(i)}\ \beta_{(i)}\ = X^t_{train(i)}y_{train(i)} \tag{19}$$

where $X_{train(i)}$ and $X_{test(i)}$ are defined on figure 10.

When we perform a n-fold-cross-validation, we must compute n different models $\beta_{(i)}, \quad i = 1, \ldots, n$ using equation 19. The only time-consuming part inside equation 19 is the computation of $X^t_{train(i)} X_{train(i)}$. Here is a small "trick" to be able to compute $X^t_{train(i)} X_{train(i)}, i = 1, \ldots, n$ in a very efficient way: If we define B_i as in figure 10, we can compute $C_i = B_i^t B_i, \quad C_i \in \Re^{n \times n}$. Then, we have:

$$\begin{array}{rrcccccc} X^t_{train(1)} & X_{train(1)} & = & C_1+ & C_2+ & C_3+ & C_4 & \\ X^t_{train(2)} & X_{train(2)} & = & C_1+ & C_2+ & C_3+ & & C_5 \\ X^t_{train(3)} & X_{train(3)} & = & C_1+ & C_2+ & & C_4+ & C_5 \\ X^t_{train(4)} & X_{train(4)} & = & C_1+ & & C_3+ & C_4+ & C_5 \\ X^t_{train(5)} & X_{train(5)} & = & & C_2+ & C_3+ & C_4+ & C_5 \\ X^t_{train} & X_{train} & = & C_1+ & C_2+ & C_3+ & C_4+ & C_5 \end{array} \tag{20}$$

Thanks to equations 20, we can compute $X^t_{train(i)} X_{train(i)}, i = 1, \ldots, n$ very easily: only a few matrix additions are needed. The same property of addition exist for the right hand side of equation 19: $X^t_{train(i)} y_{train(i)}$. Since solving equation 19 is instantaneous on modern CPU's (only the computation of $X^t_{train} X_{train}$ is time-consuming), we can easily obtain the 5 different models $\beta_{(i)}$ and the n-fold-cross-validation is "given nearly for free".

The same technique can be applied to increase the size of the validation set: see figure 11.

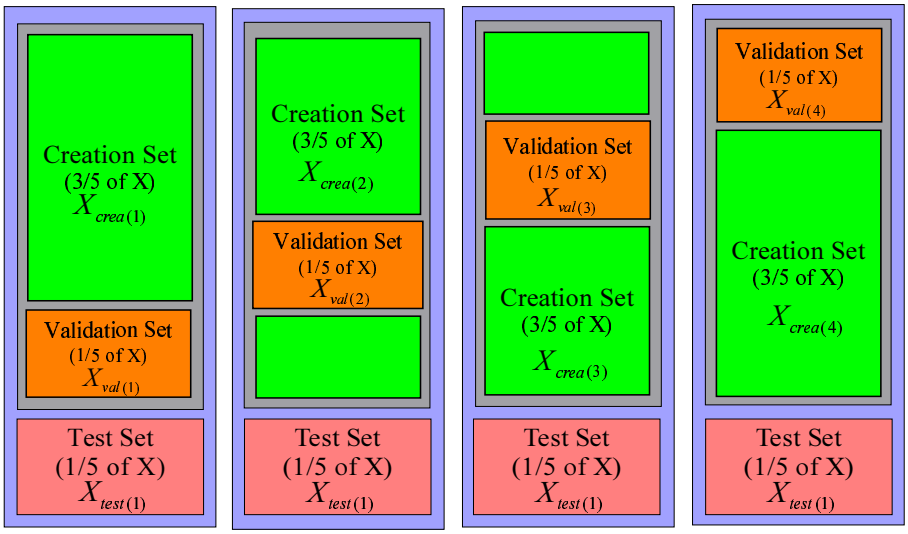

Figure 11: 4 validation sets (and 4 creation sets))

When building a model, the first step is the normalization of all the variables (the columns X_i). The normalization of a variable X_i consist of two steps:

1. We subtract from the column X_i its mean $mean(X_i)$
2. We divide the column X_i by its standard deviation $std(X_i)$

Thereafter the $C_i = B_i^t Bi$ are computed: see equation 20 and figure 10. Based on the C_i's we can compute easily many different models $\beta_{(i)}$. Let's assume that we want a model that is completely

ignoring the block B_1: the block B_1 will be used as test set. Referring to figure 10, the test set is $X_{test(5)}$ and the training set is $X_{train(5)}$. The columns in block B_2 have been normalized using $mean(X_i)$ and $std(X_i)$. The two parameters $mean(X_i)$ and $std(X_i)$ are partially based on information contained inside the block B_1. We are breaking the rule that says:

"never use any information from the test set to build a model" (21)

The columns in B_2 (part of the creation set) are normalized using information from B_1 (the test set). This small "infraction" of the rules is usually negligible and allows us to reduce considerably the computation time. However, for some rare variables, the rule must be respected strictly. These variables are named "dynamic variables". The LARS library does a strong distinction between "dynamic variables" and "static variables". The LARS library handles both types of variables. However only the "static variables" are reviewed inside this documentation.

The concept of "dynamic variables" is linked with the concept of "variable recoding". The normalization of the variables is a (very basic) recoding. If there is no recoding, there is no "dynamic variables". The "normalization of a variable" is a recoding that never produces any "dynamic variables" (especially if all the lines of X have been correctly scrambled). "Dynamic variables" appears mostly when using advanced recodings such as, for example, in the paper "1-Dimensional Splines as Building Blocks for Improving Accuracy of Risk Outcomes Models" by David S. Vogel and Morgan C. Wang. The LARS library is handling advanced recodings through the use of "dynamic variables".

Inside the LARS library, the λ parameter of the Ridge Regression is computed using a n-fold-cross-validation on the validation sets. Equations 20 are used to achieve high efficiency. The $C_i \in \Re^{n_a \times n_a}$ matrices are built incrementally each time a new variable enters the active set $\mathcal{A}$. The equations 19 are solved using an advanced Cholesky factorization that is using an advanced pivoting strategy and an advanced perturbation strategy to obtain a very high accuracy model even when the matrices $(X^t_{\cdot(i)} X_{\cdot(i)})$ are close to singularity or badly conditioned.

The LARS library has been written to use the less memory possible. For example, all the C_i matrices are symmetric, thus we only need to store half of these matrices in memory. "In place" factorization of matrices have been used when possible.

It can happen inside the LARS forward stepwise algorithm that several variables should *enter* the active set $\mathcal{A}_k$ at the same iteration k (there are several values of j that are possible inside equation 11). It can also happens that several variables should *leave* the active set $\mathcal{A}_k$ at the same iteration k. Most algorithms do not treat correctly these special cases. The LARS library is handling correctly these cases.

How to use the library?

The LARS library inside Matlab

The usage is:

```
[betaBest,betas,errors,cHats]=matlabLARS(X,y,stoppingCriteriaType,...
                            stoppingCriteriaValue,lassoModif, verbosity);
```

`matlabLARS` is an iterative algorithm that compute different β's $\in \Re^n$ such that $(X\beta - y)^2$ is minimum (classical least-square). The `X` matrix and the `y` vector do not need to be normalized.

At the first iteration only one β_i is non-null. At each iteration k, we will:

- set one more β_i to a non-null value.
- compute `Error`^{2} where `Error`$= (X\beta - y)$
- compute `cHat`$= a(k)$ (see equation 13 about $a(k)$)

The different β's, computed at each iterations, are stored inside the matrix '`betas`'. The best model among all the models according to the selected stopping criteria is stored in '`betaBest`'. In a similar way, at each iteration, the squared error `Error`^{2} and the maximum correlation values `cHat`, are stored respectively inside the matrices '`errors`' and '`cHats`'.

If the lasso modification is used (set `lassoModif`=1), then some β_i could be reset to a null value at some rare iterations.

Let's define `cHat(k)` the value of `cHat` at iteration k.
Let's define `sqEr(k)` the value of `Error`^{2} at iteration k.
Let's define `n(k)` the number of non-null β_i at iteration k.
The variable '`stoppingCriteriaType`' is a character that can be:

- 'R': if `((cHat(k-10)-cHat(k))< stoppingCriteriaValue cHat(0))`, then stop
- 'S': if `((sqEr(k-10)-sqEr(k))< stoppingCriteriaValue sqEr(0))`, then stop
- 'C': Cp test: Let's define
  - `Cp(k) = sqEr(k)-nCol+2*n(k)`
  - `p =stoppingCriteriaValue`

  if `(Cp(k)> max Cp(k-p), Cp(k-p+1),..., Cp(k-1))`, then stop
- 'M': Let's define `minCVError(k)` as the 6-Fold-Cross-Validation-Error of a model `beta(k)` that has been regularized using an optimized ridge parameter λ.
  if `(minCVError(k)>minCVError(k-1))`, then stop
- 'E': if (`stoppingCriteriaValue`=-1) include all columns, otherwise:
  if (number of non-null β_i=`stoppingCriteriaValue`), then stop
  The variable '`verbosity`' is optional. It's a number between 0 and 2. (2=maximum verbosity).

As previously mentioned, if you select a specific set of active variable, you can compute at least two different models β:

- the most efficient one (stored inside `betaBest`)
- the one used inside the LARS algorithm to search for the next variables to add inside the active set (stored inside `betas`).

`betaBest`$_{n+1}$ is the constant term of the model. All the models (in `betas` and `betaBest`) are computed to be applied on un-normalized data.

The Matlab interface to the LARS library is very poor compared to the C++ interface. The source code of the matlab interface (in C++) is inside the zip-file downloadable inside this webpage. If you want to customize the Matlab version of the LARS library, it's very easy to edit/modify the small C++ code. The C++ interface is documented below.

The LARS library as a stand-alone executable

Inside the zip-file downloadable inside this webpage, you will find the source code of a small C++ executable that solves the 'diabetes' example. The 'diabetes' data are stored inside an external *.csv* file. The last column of the *.csv* file is the target. A very fast *.csv*-file-reader is provided.

It's very easy to edit/modify the small C++ code to customize it to your needs. The C++ interface is documented below.

The LARS library in C++

The LARS library is available as a *.dll* file. To use this *.dll* file in your own project you must:

- In the "debug" version of you program, add inside the "Linker/Input/Additional Dependencies" tab: "larsDLLDebug.lib"
- Inside the directory containing the executable of the "debug" version of you program, add the file "larsDLLDebug.dll"
- In the "release" version of you program, add inside the "Linker/Input/Additional Dependencies" tab: "larsDLL.lib"
- Inside the directory containing the executable of the "release" version of you program, add the file "larsDLL.dll"
- Add inside your project the files "lars.h" and "miniSSEL1BLAS.h"

In order to work, the LARS library must access the X matrix and the y vector. The normalization of X and y is done automatically inside the LARS library, if needed. You can provide the data (X and y) in two different ways:

- ***Line by Line:*** You must create a child of the `LARS` class. Inside the child class, you must re-define some of the following functions:
  - `larsFLOAT *getLineX(int i, larsFLOAT *b);`
    This function return the i^{th} line of X. The required line can be stored inside the `b` array or inside an other array. A pointer to the array containing the line must be returned.
  - `larsFLOAT getLineY(int i);`
    This function return the i^{th} component of y: y_i. Alternatively, you can use the third parameter of the `getModel` function to specify a target.

  - `void setSelectedColumns(int n,int *s);`
    The re-definition of this function is optional. The default implementation is:

    ```
    void LARS::setSelectedColumns(int n,int *s) { selectedC=s; nSelected=n; }
    ```

    The `setSelectedColumns` function announces that subsequent calls to the function `getLineXRequired` will return only a part of the column of X. The indexes of the columns that should be returned are given inside the array `s` of length `n`.

  - `larsFLOAT *getLineXRequired(int i, larsFLOAT *b);`
    The re-definition of this function is optional. However, it's strongly recommended that you re-define this function for performance reasons. The default implementation is:

    ```
    larsFLOAT *LARS::getLineXRequired(int i, larsFLOAT *b)
    {
        int j,ns=nSelected, *s=selectedC;
        larsFLOAT *f=getLineX(i,b);
        for (j=0; j<ns; j++) b[j]=f[s[j]];
        return b;
    }
    ```

    This function returns only the columns of line `i` that have been selected using the function `setSelectedColumns`.

- ***Column by Column:*** You must create a child of the `LARS` class. Inside the child class, you must re-define the following functions:

  - `larsFLOAT *getColX(int i, larsFLOAT *b);`
    This function return the i^{th} column of X. The required column can be stored inside the `b` array or inside an other array. A pointer to the array containing the line must be returned.

  - `larsFLOAT *getColY(larsFLOAT *b);`
    This function return the target y. The target y can be stored inside the `b` array or inside an other array. A pointer to the array containing y must be returned. Alternatively, you can use the third parameter of the `getModel` function to specify a target.

Let's now assume that you have created a child of the base `LARS` class named `LARSChild`. You can now configure the different parameters of the LARS algorithm. Typically, you define these parameters inside the *constructor* of `LARSChild`. The parameters are:

- `nCol:` the number of column of X.
- `nRow:` the number of rows of X.
- `orientation:` if `orientation='L'`, then we access the X matrix line by line. Otherwise, we access the X matrix column by column.
- `lasso:` if `lasso=1`, then the lasso modification of the LARS algorithm is used.

- `fullModelInTwoSteps:` if `fullModelInTwoSteps=1`, then no forward stepwise is performed: we are directly building a model using all the column of X. When the number of column `nCol` is small this is faster than the full LARS algorithm.
- `ridge:` if `ridge=1`, then an optimization of the λ parameter of the ridge regression is performed at the end of the build of the model (this requires $nFold > 1$).
- `memoryAligned:` all the linear algebra routines are SSE2 optimized. This means that the functions `getCol*` and `getLine*` should return a pointer to an array that is 16-byte-memory-aligned. Such an array ca be obtained using the `mallocAligned` function of the *miniSSEL1BLAS* library. If you are not able to provide aligned-pointers, you can set `memoryAligned=0`, but the computing time will be more or less doubled.
- `nMaxVarInModel:` maximum number of variables inside the model when performing a forward stepwise (this is a stopping criteria).
- `nFold:` the number of block B_i (as defined in figure 10) for the n-fold-cross-validation.
- `folderBounds:` an array of size $nFold+1$ of integer that contains the line-index of the first line of each B_i (as defined in figure 10). We should also have `folderBounds[nFold]=nRow`. If you don't initialize this parameter, then `folderBounds` will be initialized for you automatically so that all the blocks B_i have the same size.
- `stoppingCriteria:` define which kind of stopping criteria will be used inside the forward stepwise:
  - `stoppingCriteria='R':` stop based on the maximum correlation a_j of the inactive columns with the residual error. see equation 13 with $s_a :=$ `stopSEP`.
  - `stoppingCriteria='S':` stop based on the L2-norm of the residual error $\|E_k\|_2$: see equation 14 with $s_E :=$ `stopSEP`.
  - `stoppingCriteria='C':` stop based on the C_p criterion: see equation 15 with $s_c :=$ `stopSEP`.
  - `stoppingCriteria='M':` stop based on the n-Fold-cross-validation-Error of the model: see equation 16.

Further configuration of the stopping criteria is available through the re-definition inside the class `LARSChild` of these two functions:

- `void initStoppingCriteria(larsFLOAT squaredError, larsFLOAT cHat);`
  This function is used to initialize internal variables for the stopping criteria.
- `char isStoppingCriteriaTrue(larsFLOAT squaredError, larsFLOAT cHat, larsFLOAT *vBetaC, LARSSetSimple *sBest);`
  If this function returns '1', the LARS forward stepwise algorithm stops. If this function returns '0', the LARS algorithm is continuing. This function update the object `sBest`. Typically, when a good selection of variable is found we do: `sBest->copyFrom(sA);`. The final model contains only the variables inside `sBest`.

This is all for the configuration of the LARS library! Now we are describing the outputs. Before using any of the outputs, you must initialize and compute many intermediate results. This is done through the function `getModel` that is usually called on the last line of the constructor of

the `LARSChild` class. The prototype of the `getModel` function is:

```
larsFLOAT *getModel(larsFLOAT **vBetaf=NULL, LARSSetSimple *sForce=NULL, LARSFloat
*target=NULL, LARSError *errorCode=NULL);
```

You usually call it this way:

```
larsFLOAT *model=NULL;
LARSSETSimple sForce(1); sForce.add(3);
getModel(&model,&sForce);
```

This means that the LARS library will construct a model β that will be returned inside the array pointed by `model` (you must free this array yourself). The model will be "forced" to use the column 3. The `LARSSETSimple` class is a class that contains a set of indexes of columns of X.

Here is a detailed description of the parameters of `getmodel`:

- **First parameter [output]:** This parameter describes an array that will be used to store the model. In the example above `model` was set to NULL, so the `getModel` function allocates itself the memory needed. You can allocate yourself the memory used to store the model: for example:
  `larsFLOAT *model=(larsFLOAT*)mallocAligned((nCol+1)*sizeof(larsFLOAT));`
  `getModel(&model);`
- **Second parameter [input]:** Describes the set of variables that will be added inside the model. If you use the forward stepwise algorithm and if you stop prematurely, this set of variable will be added to the active variables. Of course, adding a variable that is already active will have no effect.
- **Third parameter [input]:** Describes the target. This should be an array allocated with:
  `larsFLOAT *target=(larsFLOAT*)mallocAligned(nRow*sizeof(larsFLOAT)+32);`
  The LARS library will free this array for you at the right time. If you use this parameter and if (`stoppingCriteria<>'M'`), then the LARS library will not use the `getColY` or the `getLineY` functions (you don't have to define them).
- **Fourth parameter [output]:** an error code that can be interpreted using the `getErrorMessage` function.

Here is a detailed description of the (some of the) outputs:

- `sA:` you don't have direct access to the content of `sA`. Usually, you do the following:
  `getModel();`
  `LARSSETSimple sInModel(sA);`
  The object `sInModel` contains the set of indexes of all the columns that are used inside the final model.
- `invStdDev:` this is an array that contains the inverse of the standard deviation of all the columns of X. If you have normalized yourself the column of X (so that $\|X_i\|_2 = 1$), this array is `NULL`.
- `invStdDevY:` the inverse of the standard deviation of y.

- `mean:` this is an array that contains the mean of all the column of X. If you have normalized yourself the column of X (so that mean of $X_i = 0$), this array is `NULL`.
- `meanY:` the mean of y.
- `lambda:` the optimal ridge parameter λ.
- `target:` the residual error $E_k \in \Re^m$ at the end of the LARS algorithm (NOT normalized).

Here is a small sketch of a backward stepwise algorithm:

1. Build a model using the `getmodel` function. Stop prematurely. The index of the active variables are stored inside `sA`.
2. Try to remove variable i:
   - Get a copy `sCopy` of all the variable inside the model: `sCopy.copyrom(sA);`
   - Remove variable i from `sCopy`: `sCopy.remove(i);`. Let's define `na`, the number of variable inside `sCopy`: `int na=sCopy.n;`.
   - This computes a model using only the columns inside `sCopy`:
     `larsFLOAT *vBeta=(larsFLOAT*)malloc((nCol+1)*sizeof(larsFLOAT));`
     `buildModel(vBeta, lambda, -1, &sCopy);`
     A model in "compressed/short form" is stored inside the array `vBeta`. `vBeta` is currently an array of size `na`. `vBeta[j]` is the weight of the model given to the column of index `sCopy.s[j]` The short form is a very efficient way of storing the model when combined with functions such as `getLineXRequired`. To convert the model to the "normal/long form", use: `shortToLongModel(vBeta);` Note that the size of the "long form" of the model is `nCol+1` because we need also to store the offset(constant term) of the model. Inside this example the model is built using all the blocks B_i (see figure 10 for a definition of the B_i's). If we want to build a model using all the blocks B_i except the block B_4, we will do: `buildModel(vBeta, lambda, 4, &sCopy);` This allows us to perform easily a n-fold-cross-validation-error test.
   - Test the model to see if we can remove column i without having a loss of efficiency. This test may involves a n-fold-cross-validation technique. If this is the case, we can drop definitively the column i: we do: `keepVars(sCopy);` (sA will be updated).
   - choose another column i inside `sA` and go back to the first point of step 2.

As illustrated above, thanks the `buildModel` function, we can obtain easily up to `nFold` different models based on the same set `sA` of variables. Using the function `buildAverageModel`, we can compute one last model $\beta_{average}$ that is the *average* of these `nFold` different models. This last model $\beta_{average}$ is a little bit less performant than the other models but it is a lot more stable.

To summarize, there is at least `nFold+3` different models based on the same set `sA` of variables:

- The model that is produced by the forward stepwise LARS algorithm when searching for informative variables inside the stepwise iterations. This model is suboptimal (see remark about "greedy" algorithms to know why). You can access this model through the `isStoppingCriteriaTrue` function.

- The model given by the function `buildModel(·,·,−1)`: this model is using all the blocks B_i inside the creation set.
- The models given by the function `buildModel(·,·,i)`: these models are using all the the creation set minus the block B_i. There are `nFold` models of this type ($i = 0, \ldots,$`nFold`-1).
- The model given by the function `buildAverageModel` that is an average of the `nFold` models computed using a n-fold-cross-validation technique.

If you give $\lambda = 0$ as second parameter of the `buildModel` function, no model stabilization through the ridge technique is performed. If you give $\lambda^* =$`lambda` as second parameter of `buildModel`, the optimal λ^* value stored in `lambda` is used to stabilize the model. Each time you remove a variable from the model, you should recompute $\lambda^* =$`lambda`: this is done through the function `minimalCrossValError`.

Here are some more useful functions:

- `larsFLOAT normalizedLinearCorrelation(int i, int j, int cv=-1);`
  This will return C_{ij}: the linear correlation between variable i and j: $C_{ij} = X_i^t X_j$. The result is normalized so that $0 \leq C_{ij} \leq 1$. If `cv=4`, then the block B_4 is ignored while computing C_{ij}.
- `larsFLOAT linearCorrelationWithTarget(int i, int cv=-1);`
  This will return a_i: the linear correlation between variable i and the target y: $a_i = X_i^t y$. The result is normalized so that $0 \leq a_i \leq 1$. If `cv=4`, then the block B_4 is ignored while computing a_i.
- `static larsFLOAT apply(int n, larsFLOAT *line, larsFLOAT *model);`
  This computes $\sum_{i=0}^{i<n} line[i] * model[i] + model[n]$ (usually `n=nCol`).

You get two examples of use when you download the LARS Library:

1. The source code of a matlab mex-file function based on the LARS library. This allows you to use the LARS Library under Matlab. Due to limitation of Matlab, we have to set the options `orientation='C'` and `memoryAlignment=0`. Thus, the library will be rather slow compared to a full C++ implementation. The classical 'diabetes' example is given (the same one than inside the main paper on LARS). An evolution of the example example illustrated in figure 8 is also given. This example is interesting because the FOS algorithm fails on it. The final model of the FOS algorithm contains only one variable: the X_3 variable. On the contrary, the LARS algorithm is able to find the right set of active variables: X_1 and X_2.
2. The source code of a small C++ executable that solves the 'diabetes' example. The 'diabetes' data are stored inside an external *.csv* file. The last column of the *.csv* file is the target. A very fast *.csv*-file-reader is provided.

About the LARS library

The functionalities of this library are still not completely documented. There are many other functionalities linked to the correct usage of a "rotating" validation set ("rotating" means a

n-fold-cross-validation validation set). Currently, the documentation covers only the case of a "rotating" test set.

The documentation does not cover the "dynamic variables" (see equation 21 about "dynamic variables").

The library allows you to construct second order, third order,... models very easily.

If `memoryAlignment=1`, all the matrix algebra are performed using an optimized SSE2 library. If `orientation='L'` and `memoryAlignment=1`, the computation time is divided by two compared to `memoryAlignment=0`. In this case, it's important to store the data "line by line" otherwise we lose the "locality" of the computations. "Locality" means that all the data needed to perform a large part of the computations are inside the (small) cache of the processor. From a memory consumption standpoint, the "line by line" mode is better. For all these reasons, the "line by line" mode is preferable to the "column by column" mode.

Inside the LARS algorithm, the Cholesky factorizations are performed incrementally as new variables enter the model. The Cholesky factorizations are based on the paper "A revised Modified Cholesky Factorization Algorithm" by Robert B. Schnabel and Elisabeth Eskow. This advanced Cholesky Factorization allows us to obtain high accuracy models even when close to rank deficiency.

The optimization of the ridge parameter λ is performed using a modified Brent algorithm to ensure global convergence. The optimization algorithm has been extracted from an optimization library named CONDOR.

References

Some papers about the Fast-Orthogonal-Search:

- An introductory paper: "Application of Fast Orthogonal Search for the Design of RBFNN", W. Ahmed, D.M. Hummels and M.T. Musavi
- A complete thesis: "Fast Orthogonal Search For Training of Radial Basis Function Neural Networks" by Wahid Ahmed
- An example why LARS is better than LASSO for ARMA models: "A new Algorithm for Linear and non-linear ARMA Model Parameter Estimation Using Affine Geometry", Sheng Lu, Ki Hwan Ju and Ki H. Chon.
- Some application of FOS:
  - "Orthogonal Approaches to Time-series Analysis and System Identification", Michael J. Korenberg.
  - "Spectral Analysis of Heart Valve Sound for detection of Prosthetic Heart Valve Diseases", Sang Huyn Kim, Hee Jong Lee, Jae Man Huh and Byung Chul Chang

An easy, experimental and fun paper about model stabilization via Ridge: "The *refmix* dataSet", Mark J.L. Orr.

A paper about Lasso: "Regression shrinkage and selection via the lasso", Robert Tibshirani

A paper about LARS: "Least Angle Regression", Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani

Cholesky factorization is based on the paper "A revised Modified Cholesky Factorization Algorithm" by Robert B. Schnabel and Elisabeth Eskow.

A very good advanced recoding: "1-Dimensional Splines as Building Blocks for Improving Accuracy of Risk Outcomes Models" by David S. Vogel and Morgan C. Wang.