Journal of Statistical Software

October 2018, Volume 87, Issue 3. doi: 10.18637/jss.v087.i03

Simulated Data for Linear Regression with Structured and Sparse Penalties: Introducing pylearn-simulate

Tommy Löfstedt
Neurospin, CEA Saclay

Vincent Guillemot
Neurospin, CEA Saclay

Vincent Frouin
Neurospin, CEA Saclay

Edouard Duchesnay
Neurospin, CEA Saclay

Fouad Hadj-Selem
ITE Institute: VeDeCoM

Abstract

A currently very active field of research is how to incorporate structure and prior knowledge in machine learning methods. It has lead to numerous developments in the field of non-smooth convex minimization. With recently developed methods it is possible to perform an analysis in which the computed model can be linked to a given structure of the data and simultaneously do variable selection to find a few important features in the data. However, there is still no way to unambiguously simulate data to test proposed algorithms, since the exact solutions to such problems are unknown.

The main aim of this paper is to present a theoretical framework for generating simulated data. These simulated data are appropriate when comparing optimization algorithms in the context of linear regression problems with sparse and structured penalties. Additionally, this approach allows the user to control the signal-to-noise ratio, the correlation structure of the data and the optimization problem to which they are the solution.

The traditional approach is to simulate random data without taking into account the actual model that will be fit to the data. But when using such an approach it is not possible to know the exact solution of the underlying optimization problem. With our contribution, it is possible to know the exact theoretical solution of a penalized linear regression problem, and it is thus possible to compare algorithms without the need to use, e.g., cross-validation.

We also present our implementation, the Python package **pylearn-simulate**, available at `https://github.com/neurospin/pylearn-simulate` and released under the BSD 3-clause license. We describe the package and give examples at the end of the paper.

Keywords: simulated data, sparse and structured penalties, linear regression, Python.

1. Introduction

Simulated data are widely used to assess optimization methods. This is because of their ability to evaluate certain aspects of the methods under study; and these aspects are impossible to look into when using real data sets. In the context of convex optimization, it is never possible to know the exact solution of the minimization problem with real data and it is a difficult problem even with simulated data. We propose to generalize an approach originally published by Nesterov (2013), for LASSO regression, to a broader family of penalized regression problems.

We would like to generate simulated data for which we know the exact solution of a given function. The inputs are: the minimizer β^* ($p \times 1$), a candidate data set X_0 ($n \times p$), a residual vector ε ($n \times 1$), regularization parameters (in our case there are up to three of these), the signal-to-noise ratio σ, and the expression of a function, $f(\beta)$, that is to be minimized.

The candidate version of the data set may for instance be $X_0 \sim N(\mu, \Sigma)$, and the residual vector may be $\varepsilon \sim N(0, 1)$. X_0 can alternatively be a data set relevant for the application in mind: a set of images, microarrays, etc. Using real data as input could provide a more realistic correlation structure, and this correlation structure is not affected by our method.

The procedure proposed here outputs X and y such that

$$\beta^* = \arg\min_{\beta} f(\beta), \tag{1}$$

with f a convex function of β that depends on the parameters that define the simulated data, (X_0, β, ε, σ), and on the regularization parameters. In a nutshell, X is obtained by scaling each column $j = 1, \ldots, p$ of X_0 by a factor ω_j. We give the expression of ω_j as a function of the inputs and the subdifferential of f in Section 3.

In linear regression, simulated data are often generated such that

$$y = X\beta^* + \varepsilon. \tag{2}$$

If we want to evaluate an algorithm to minimize a penalized regression problem

$$f(\beta) = \frac{1}{2}\|X\beta - y\|_2^2 + P(\beta), \tag{3}$$

then we need to use, e.g., cross-validation to find an acceptable value for any regularization parameter that could be hidden in the penalty P. But the found values are very likely suboptimal, and in any case, we are forced to compare the solution to Equation 2.

Our contribution is thus to give the solution that actually minimizes Equation 3, instead of Equation 2, namely β^* from Equation 1. This means we are able to compare speed, sensitivity to noise, correlation, etc., and the actual solutions found by different minimization algorithms.

We present an implementation of the procedures presented below in an object oriented Python (van Rossum *et al.* 2011) package, **pylearn-simulate**, with which it is straight-forward to generate such data (Löfstedt, Guillemot, Frouin, Duchesnay, and Hadj-Selem 2014a). **pylearn-simulate** is developed for Python 2.7.x and is available at `https://github.com/neurospin/pylearn-simulate`.

An R markdown document (`Example_in_R.rmd`) is also provided as a supplementary file in order to illustrate to the reader familiar with R (R Core Team 2018) the simplicity of our

method and its impact on a realistic gene expression data set. This R markdown example was developed using R version 3.1.1 and RStudio version 0.98.1062 (**RStudio** 2013).

2. Installation

pylearn-simulate requires some external packages in order to work. They are, however, very few, and are freely available online.

- **NumPy** version 1.6.1 or newer. This is the fundamental package for scientific computing in Python and contains, among other things, linear algebra functions and matrix/vector objects (van der Walt, Colbert, and Varoquaux 2011).

- **SciPy** version 0.9.0 or newer. A library of open-source software for mathematics, science and engineering (Jones, Oliphant, and Peterson 2001).

- **pylearn-parsimony** version 0.2.1 or newer; optional. A library for structured and sparse machine learning. Contains several algorithms for minimizing loss functions with complex structured penalties. While optional, the examples require this package in order to run fully (Löfstedt, Guillemot, Hadj-Selem, Li, Frouin, and Duchesnay 2014b).

Once the dependencies are installed, **pylearn-simulate** can be installed by obtaining a release version from `https://github.com/neurospin/pylearn-simulate`. Unpack the file, go to the **pylearn-simulate** directory and type:

```
$ python setup.py install --user
```

for a local installation in the user's *userbase* directory (usually in `~/.local/lib/python2.7/-site-packages` on Unix-like/-based operating systems, such as Linux and OS X, and in `%AppData%\Python\Python27\site-packages` on Windows), or

```
$ sudo python setup.py install
```

for a global installation accessible to all users. You will need to have administrator rights on your computer in order to install software for all users.

3. Notations

We place ourselves in the context of linear regression models. Let $X \in \mathbb{R}^{n \times p}$ be a matrix of n samples, where each sample lies in a p-dimensional space; and let $y \in \mathbb{R}^n$ denote the n-dimensional response vector. The p-vector that contains the regression coefficients is denoted β. In the following, we denote by $\|\cdot\|_q$ the standard ℓ_q-norm on $\mathbb{R}^p$ with dual norm $\|\cdot\|_{q'}$.

For a smooth real function f, we denote by $\nabla f(\beta)$ its gradient. Finally, the subdifferential, i.e. the set of subgradients, of a convex function f is denoted $\partial f(\beta)$.

4. Method

The objective is to generate X and y such that

$$\beta^* = \arg\min_{\beta} \frac{1}{2}\|X\beta - y\|_2^2 + P(\beta), \tag{4}$$

where P is a penalty that can be expressed in the form

$$P(\beta) = \sum_{\pi \in \Pi} \lambda_\pi \pi(\beta),$$

in which Π is the set of all our penalties, $\lambda_\pi > 0$ is the regularization parameter associated with penalty π, and $\partial\pi(\beta^*)$, used below, is the subdifferential of penalty π at β^*. This is a general notation to represent the fact that we may have several different penalties.

If the subdifferential of each penalty is known, the scaling factors, ω_j, that we are looking for can each be expressed as the jth component of $\sum_{\pi\in\Pi} -\lambda_\pi \partial\pi(\beta^*)$ divided by $X_{0,j}^\top \varepsilon$. More details are given below.

The penalties that we consider in the examples in this work are

$$P(\beta) = \frac{\lambda_{\ell_2}}{2}\|\beta\|_2^2 + \lambda_{\ell_1}\|\beta\|_1 + \lambda_{\text{TV}}\text{TV}(\beta),$$

where TV is the total variation function (Rudin, Osher, Fatemi, and Monica 1992), and

$$P(\beta) = \frac{\lambda_{\ell_2}}{2}\|\beta\|_2^2 + \lambda_{\ell_1}\|\beta\|_1 + \lambda_{\text{GL}}\text{GL}(\beta),$$

where GL is the overlapping group LASSO penalty (Yuan and Lin 2006) and, as we show below, we know their subdifferential.

4.1. Algorithm

Nesterov (2013) addressed how to simulate data for the simpler case of the LASSO regression, and we will therefore not go into the details here. The algorithm given in Nesterov (2013) is adapted to our notations and is detailed in Appendix A.

The principle behind Nesterov's idea, that we generalize for complex penalties, is as follows: First, define the residual to be $\varepsilon = X\beta - y$, in the model between $X\beta$ and y, such that it is independent from β^*. Then, select acceptable values for the columns of X such that zero belongs to the subdifferential of f at point β^*. It requires then to have a first unscaled version of the dataset, X_0, whose columns $X_{0,j}$ $(j = 1, \ldots, p)$ are each scaled by a factor ω_j.

The algorithm is detailed in Algorithm 1. This algorithm is used to generate a simulated data set that is the solution to a complex optimization problem.

The vector r_π belongs to the subdifferential of penalty π at β^*. We chose to generate such vectors randomly in our implementation, but it is up to the user to generate them according to certain additional constraints on the obtained data set. When X_0 is a real-world data set, the purpose of such constraints could be to keep certain properties that are not directly in relation to the optimization problem. For example, when transforming images, certain desirable shapes have to remain visible in the resulting data set.

Algorithm 1 Generate a simulated data set.

Require: β^*, X_0, ε
Ensure: X and y such that $\beta^* = \arg\min_\beta f(\beta)$
1: **for** $\pi \in \Pi$ **do**
2: Generate $r_\pi \in \partial\pi(\beta^*)$
3: **end for**
4: **for** $i = 1, \ldots, p$ **do**
5: $\omega_j = \frac{-\sum_{\pi\in\Pi} \lambda_\pi r_{\pi,j}}{X_{0,j}^\top \varepsilon}$
6: $X_j = \omega_j X_{0,j}$
7: **end for**
8: $y = X\beta^* - \varepsilon$

While this algorithm is fairly general, it can only be used when the subdifferential of f is explicit. We show in the following that it is possible to have an analytical expression for a wide variety of complex convex penalties.

4.2. Subdifferential of complex penalties

The complex penalties that we consider in this work can be written in the form

$$\pi(\beta) = \sum_{g=1}^{G} \|A_g\beta\|_q. \tag{5}$$

While any q-norm is possible, we will in this work only be interested in the case when $q = q' = 2$, i.e., the Euclidean norm. This is the case when π is, e.g., the total variation or (overlapping) group LASSO penalties.

We need the following two lemmas in order to derive the subdifferential of the complex penalties.

Lemma 4.1 (Subdifferential of the sum)**.** *If f_1 and f_2 are convex functions with domain $\mathbb{R}^p$, then*

$$\partial(f_1 + f_2) = \partial f_1 + \partial f_2.$$

Proof. See Theorem 4.1.1 on Page 183 in Hiriart-Urrut and Lemaréchal (2004). □

Lemma 4.2 (Subdifferential of the composition)**.** *If f is convex and defined on the image space of A and $g(x) = Ax$ is a linear function with domain $\mathbb{R}^p$, then*

$$\partial(f \circ g)(x) = A^\top \partial f(g(x)) = A^\top \partial f(Ax).$$

Proof. See Theorem 4.2.1 on Page 184 in Hiriart-Urrut and Lemaréchal (2004). □

These lemmas play a central role in the following theorem that details the structure of the subdifferential of π.

Theorem 4.3 (Subdifferential of π)**.** *If π has the form given in Equation 5, then*

$$\partial\pi(\beta) = A^\top \begin{bmatrix} \partial\|A_1\beta\|_2 \\ \vdots \\ \partial\|A_G\beta\|_2 \end{bmatrix}.$$

Proof.

$$\begin{aligned}
\partial\pi(\beta) &= \partial\left(\sum_{g=1}^{G}\|A_g\beta\|_2\right) \\
&= \sum_{g=1}^{G}\partial\|A_g\beta\|_2 && \text{(Using Lemma 4.1)} \\
&= \sum_{g=1}^{G}A_g^\top\partial\|A_g\beta\|_2 && \text{(Using Lemma 4.2)} \\
&= A^\top\begin{bmatrix} \partial\|A_1\beta\|_2 \\ \vdots \\ \partial\|A_G\beta\|_2 \end{bmatrix},
\end{aligned}$$

where

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_G \end{bmatrix}.$$

□

Before we show the application to some actual penalties we will mention that the subdifferential of the ℓ_2-norm is

$$\partial\ell_2(x) = \partial\|x\|_2 = \begin{cases} \left\{\frac{x}{\|x\|_2}\right\} & \text{if } \|x\|_2 > 0, \\ \{y \mid \|y\|_2 \leq 1\} & \text{if } \|x\|_2 = 0, \end{cases} \tag{6}$$

i.e., the second case (when $\|x\|_2 = 0$) corresponds to the set of points in the unit ℓ_2 ball.

4.3. Smoothed penalties

The authors have recently published an article detailing an algorithm that utilizes a smoothing technique proposed by Nesterov (2004). The authors described an efficient minimization of a convex non-differentiable function involving the total variation penalty (Hadj-Selem, Löfstedt, Dohmatob, Frouin, Dubois, Guillemot, and Duchesnay 2018).

Nesterov's smoothing technique is a very efficient way of approximating non-smooth functions by a smoothed function. The smoothed function is regularized such that when the regularization, or smoothing, parameter approaches zero, the approximation approaches the original function. We will not describe this technique here, but refer to Nesterov (2004) or Hadj-Selem *et al.* (2018) for details.

When a complex penalty function, π, is smoothed using Nesterov's method, the smoothed function is defined as

$$\pi_\mu(\beta) = \langle \alpha^*, A\beta \rangle - \frac{\mu}{2}\|\alpha^*\|_2^2,$$

where μ is a parameter that controls the smoothing,

$$\alpha^* = \arg\max_{\alpha \in K} \left\{ \langle \alpha, A\beta \rangle - \frac{\mu}{2}\|\alpha\|_2^2 \right\},$$

the operator $\langle \cdot, \cdot \rangle$ denotes the inner product of the arguments, K is a compact convex set in a finite-dimensional vector space, and A is a linear operator that transforms between two finite-dimensional vector spaces.

The gradient of $\pi_\mu(\beta)$ is defined as

$$\nabla \pi_\mu(\beta) = A^\top \alpha^*. \tag{7}$$

It is thus straight-forward to simulate such data. The A matrices for total variation and (overlapping) group LASSO are defined in Appendix B. The smoothing, or regularization parameter μ is user-defined, but usually selected to be *small*. The computation of subgradients for TV and (overlapping) group LASSO are detailed in Appendix B.

5. Application

We apply the aforementioned algorithm to generate a data set and associate it to the exact solution of a linear regression problem with elastic net and a Nesterov-smoothed complex convex penalty.

5.1. Linear regression with elastic net and a complex penalty

We will here give an example with elastic net and a complex penalty, such as, e.g., the total variation or group LASSO penalties. The function we are working with is

$$f(\beta) = \frac{1}{2}\|X\beta - y\|_2^2 + \frac{\lambda_{\ell_2}}{2}\|\beta\|_2^2 + \lambda_{\ell_1}\|\beta\|_1 + \lambda_\pi \pi(\beta),$$

where π is a complex penalty, and λ_{ℓ_2}, λ_{ℓ_1} and λ_{ℓ_π} are regularization parameters. The subdifferential in this case has

$$0 \in \partial f(\beta) = X^\top \varepsilon + \lambda_{\ell_2}\beta + \lambda_{\ell_1}\partial\|\beta\|_1 + \lambda_\pi \partial\pi(\beta), \tag{8}$$

where ε is distributed like the residuals, $X\beta - y$. We rearrange and note that we seek

$$X_j^\top \varepsilon \in -\lambda_{\ell_2}\beta_j - \lambda_{\ell_1}\partial|\beta_j| - \lambda_\pi(\partial\pi(\beta))_j, \tag{9}$$

where we denote by $(\cdot)_j$ the jth component of the vector within parentheses. Furthermore, since we define $X_j = \omega_j X_{0,j}$, we have

$$\omega_j \in \frac{-\lambda_{\ell_2}\beta_j - \lambda_{\ell_1}\partial|\beta_j| - \lambda_\pi(\partial\pi(\beta))_j}{X_{0,j}^\top \varepsilon}. \tag{10}$$

We note that in the case when $\beta_j = 0$, adding the smooth ridge penalty to the LASSO has no effect on the generated data.

We can show that both complex penalties, i.e. total variation and group LASSO, can be formulated as in Equation 5 (detailed in Appendix B). It is then possible to use Theorem 4.3 to obtain an expression for the sub-differential of π as

$$\partial\pi(\beta) = A^\top \begin{bmatrix} \partial\|A_1\beta\|_2 \\ \vdots \\ \partial\|A_G\beta\|_2 \end{bmatrix}.$$

Replacing this expression in Equation 10, we obtain

$$\omega_j \in \frac{-\lambda_{\ell_2}\beta_j - \lambda_{\ell_1}\partial|\beta_j| - \lambda_\pi \left(A^\top \begin{bmatrix} \partial\|A_1\beta\|_2 \\ \vdots \\ \partial\|A_G\beta\|_2 \end{bmatrix} \right)_j}{X_{0,j}^\top \varepsilon}, \tag{11}$$

for each variable $j = 1, \ldots, p$ and with $X_j = \omega_j X_{0,j}$. We also note from Equation 6 that $\partial|x|$ is $\{\text{sign}(x)\}$ if $x \neq 0$ and $x \in [-1, 1]$ if $x = 0$; thus, if $x = 0$, we may choose $x \sim \mathcal{U}(-1, 1)$.

Further, if the non-smooth complex penalty, π, be substituted for a Nesterov-smoothed complex penalty, π_μ, the subdifferential of π in Equation 10 could simply be replaced by the gradient of π_μ, defined in Equation 7.

5.2. Intercept

It is very common to include an intercept term in the model, and thus instead of Equation 4 solve

$$\beta^* = \arg\min_\beta \frac{1}{2}\|X\beta + \beta_0 - y\|_2^2 + P(\beta), \tag{12}$$

where β_0 is the intercept term, and P defines all penalties. The intercept term is usually included in the original problem by extending β and X such that

$$\beta := \begin{bmatrix} \beta_0 \\ \beta \end{bmatrix}$$

and

$$X := [\mathbb{1}_n, X],$$

where $\mathbb{1}_n$ is an $n \times 1$ vector of ones.

We note that the penalties do not include the intercept term, and, therefore, that the gradient of the penalties, with respect to β_0, is zero. We also note that the intercept column may not change from X_0 to X by this procedure, and thus that $\omega_0 = 1$. We remember Equation 9 and that the intercept column is a column of ones. Then, we write Equation 9 for the intercept as

$$X_{j=0}^\top \varepsilon = \omega_0 X_{0,j=0}^\top \varepsilon = 1 \cdot \mathbb{1}_n^\top \varepsilon = 1 \cdot \sum_{i=1}^n \varepsilon_i = 0.$$

Thus, we have properly included the intercept term if the residual terms sum to zero. Therefore, to handle the intercept, we change β and X as described above and make sure that $\sum_{i=1}^{n} \varepsilon_i = 0$. If this is not the case, we make it so by subtracting the mean; i.e., we let

$$\varepsilon := \varepsilon - \frac{1}{n} \sum_{i=1}^{n} \varepsilon_i.$$

5.3. Signal-to-noise ratio

We use the same definition of signal-to-noise ratio as Bach, Jenatton, Mairal, and Obozinski (2011), namely that

$$\text{SNR} = \frac{\|X(\beta)\beta\|_2}{\|\varepsilon\|_2},$$

where $X(\beta)$ is the data generated from β when using the simulation process described above. With this definition of signal-to-noise ratio, and with the definition of the simulated data given above we may scale the regression vector such that

$$\text{SNR}(a) = \frac{\|X(\beta a)\beta a\|_2}{\|\varepsilon\|_2}. \tag{13}$$

If the user provides a desired signal-to-noise ratio, σ, it is reasonable to ask if we are able to find an a such that $\text{SNR}(a) = \sigma$. We have the following theorem.

Theorem 5.1. *Using the definition of simulated data described above, and with the definition of signal-to-noise ratio in Equation 13 there exists an $a > 0$ such that*

$$SNR(a) = \sigma, \tag{14}$$

for $\sigma > 0$, $\|\beta\|_2 > 0$, $\|X_0^\top \varepsilon\|_2 > 0$, $\|\varepsilon\|_2 < \infty$, and either $\|X_0 \widetilde{\beta}_k\|_2 > 0$ or $\|X_0 \widetilde{\beta}_m\|_2 > 0$, where $\widetilde{\beta}_k$ and $\widetilde{\beta}_m$ are defined below.

Proof. We rearrange the signal-to-noise ratio as

$$\|X(\beta a)\beta a\|_2 = \sigma \|\varepsilon\|_2, \tag{15}$$

and square both sides to get

$$\|X(\beta a)\beta a\|_2^2 = \sigma^2 \|\varepsilon\|_2^2 =: s. \tag{16}$$

We let X_j be the jth column of $X(\beta a)$, remember that $X_j = \omega_j X_{0,j}$, and let β_j be the jth element of β. The left-hand side of Equation 16 is then

$$\begin{aligned} \|X(\beta a)\beta a\|_2^2 &= \left(\sum_{j=1}^{p} X_j \beta_j a \right)^\top \left(\sum_{i=1}^{p} X_j \beta_j a \right) \\ &= \sum_{j=1}^{p} \sum_{\substack{l=1 \\ l \neq j}}^{p} a^2 X_j^\top X_l \beta_j \beta_l + \sum_{j=1}^{p} a^2 X_j^\top X_j \beta_j^2 \\ &= \sum_{j=1}^{p} \sum_{\substack{l=1 \\ l \neq j}}^{p} a^2 X_{0,j}^\top X_{0,l} \beta_j \beta_l \omega_j \omega_l + \sum_{j=1}^{p} a^2 X_{0,j}^\top X_{0,j} \beta_j^2 \omega_j^2. \end{aligned} \tag{17}$$

Using Equation 6 and $a > 0$, noting that

$$\frac{ax}{\|ax\|_2} = \frac{ax}{a\|x\|_2} = \frac{x}{\|x\|_2},$$

we conclude that $\partial_{a\beta_j}|a\beta_j| = \partial_{\beta_j}|\beta_j|$ and $(\partial_{a\beta_j}\pi(a\beta))_j = (\partial_{\beta_j}\pi(\beta))_j$, where by ∂_x we denote the subdifferential with respect to x, and hence the partial subdifferential of the ℓ_1 and complex penalties are independent of any scaling factor a.

If we add all the penalties described above, i.e., ℓ_1, ℓ_2 and a complex penalty such as TV (or GL), as has been the case throughout, and compute the partial derivatives of Equation 11 with respect to $a\beta$, we obtain

$$\begin{aligned}
\omega_j &\in \frac{-a\lambda_{\ell_2}\beta_j - \lambda_{\ell_1}\partial_{a\beta_j}|a\beta_j| - \lambda_\pi \left(A^\top \begin{bmatrix} \partial_{a\beta}\|A_1 a\beta\|_2 \\ \vdots \\ \partial_{a\beta}\|A_G a\beta\|_2 \end{bmatrix} \right)_j}{X_{0,j}^\top \varepsilon} \\
&= \frac{-a\lambda_{\ell_2}\beta_j - \lambda_{\ell_1}\partial|\beta_j| - \lambda_\pi \left(A^\top \begin{bmatrix} \partial_{\beta}\|A_1 \beta\|_2 \\ \vdots \\ \partial_{\beta}\|A_G \beta\|_2 \end{bmatrix} \right)_j}{X_{0,j}^\top \varepsilon}.
\end{aligned}$$

Hence, we may thus write ω_j as a function of a by

$$\omega_j = k_j a + m_j,$$

where

$$k_j = \frac{-\lambda_{\ell_2}\beta_j}{X_{0,j}^\top \varepsilon}$$

and

$$m_j \in \frac{-\lambda_{\ell_1}\partial|\beta_j| - \lambda_\pi \left(A^\top \begin{bmatrix} \partial\|A_1 \beta\|_2 \\ \vdots \\ \partial\|A_G \beta\|_2 \end{bmatrix} \right)_j}{X_{0,j}^\top \varepsilon}.$$

We continue to expand Equation 17 and obtain

$$\begin{aligned}
&\sum_{j=1}^{p}\sum_{\substack{l=1\\l\neq j}}^{p} a^2 X_{0,j}^\top X_{0,l}\beta_j\beta_l\omega_j\omega_l + \sum_{j=1}^{p} a^2 X_{0,j}^\top X_{0,j}\beta_j^2\omega_j^2 \\
&\quad= \sum_{j=1}^{p}\sum_{\substack{l=1\\l\neq j}}^{p} a^2 X_{0,j}^\top X_{0,l}\beta_j\beta_l(k_j a + m_j)(k_l a + m_l) + \sum_{j=1}^{p} a^2 X_{0,j}^\top X_{0,j}\beta_j^2(k_j a + m_j)^2 \\
&\quad= \sum_{j=1}^{p}\sum_{\substack{l=1\\l\neq j}}^{p} a^2 \underbrace{X_{0,j}^\top X_{0,l}\beta_j\beta_l}_{:=d_{j,l}}(a^2 k_j k_l + a k_j m_l + a m_j k_l + m_j m_l) \\
&\qquad+ \sum_{j=1}^{p} a^2 \underbrace{X_{0,j}^\top X_{0,j}\beta_j^2}_{:=d_{j,j}}(a^2 k_j^2 + 2a k_j m_j + m_j^2). \\
&\quad= \sum_{j=1}^{p}\sum_{\substack{l=1\\l\neq j}}^{p} a^4 d_{j,l}k_j k_l + a^3 d_{j,l}(k_j m_l + m_j k_l) + a^2 d_{j,l} m_j m_l \\
&\qquad+ \sum_{j=1}^{p} a^4 d_{j,j}k_j^2 + 2a^3 d_{j,j}k_j m_j + a^2 d_{j,j} m_j^2.
\end{aligned}$$

We note that this is a fourth order polynomial of a and write it in the generic form

$$\|X(\beta a)\beta a\|_2^2 = Aa^4 + Ba^3 + Ca^2. \tag{18}$$

Now, since we seek a solution $a > 0$ such that $\|X(\beta a)\beta a\|_2^2 = s > 0$, we seek positive roots of the quartic equation

$$Aa^4 + Ba^3 + Ca^2 - s = 0. \tag{19}$$

This fourth order polynomial, in the left-hand side of Equation 19, has a minimum of $-s$ when $a \to 0$, since Equation 18 is non-negative for all values of a.

We note that the coefficient for the quartic term, A, is non-negative, since

$$\begin{aligned}
A = \sum_{j=1}^{p}\sum_{\substack{l=1\\l\neq j}}^{p} d_{j,l}k_j k_l + \sum_{j=1}^{p} d_{j,j}k_j^2 &= \sum_{j=1}^{p}\sum_{l=1}^{p} d_{j,l}k_j k_l \\
&= \sum_{j=1}^{p}\sum_{l=1}^{p} k_j\beta_j X_{0,j}^\top X_{0,l}\beta_l k_l \\
&= \widetilde{\beta}_k^\top X_0^\top X_0 \widetilde{\beta}_k \\
&\geq 0,
\end{aligned}$$

where the elements of $\widetilde{\beta}_k$ are $\beta_j k_j$, for $j = 1, \ldots, p$.

In case $A = \widetilde{\beta}_k^\top X_0^\top X_0 \widetilde{\beta}_k = 0$, then $\|X_0\widetilde{\beta}_k\|_2 = 0$ and we note that B is also equal to 0, because

$$\begin{aligned} B = \sum_{j=1}^{p} \sum_{\substack{l=1 \\ l \neq j}}^{p} d_{j,l}(k_j m_l + m_j k_l) + \sum_{j=1}^{p} 2 d_{j,j} k_j m_j &= \sum_{j=1}^{p} \sum_{l=1}^{p} d_{j,l}(k_j m_l + m_j k_l) \\ &= 2 \sum_{j=1}^{p} \sum_{l=1}^{p} k_j \beta_j X_{0,j}^\top X_{0,l} \beta_l m_l \\ &= 2 \underbrace{\widetilde{\beta}_k^\top X_0^\top}_{=0} X_0 \widetilde{\beta}_m, \\ &= 0, \end{aligned}$$

where the elements of $\widetilde{\beta}_m$ are $\beta_j m_j$, for $j = 1, \ldots, p$. Hence, in order for there to be roots when $A = 0$, the coefficient of the quadratic must be positive. We note that since $s > 0$, if $A = 0$, then we must have that $C > 0$ which implies that $\|X_0\widetilde{\beta}_m\|_2 > 0$.

To sum up, if $A > 0$, the quartic equation tends to infinity when a tends to infinity. Further, if $A = 0$, then $B = 0$ and $C > 0$, which means that the equation will also tend to infinity when a tends to infinity.

Thus, by the intermediate value theorem there is a value of a for which $\|X(\beta a)\beta a\|_2^2 - s = 0$ and thus also that $\text{SNR}(a) = \sigma$. □

We may use Equation 19 above to find the roots of this fourth order polynomial analytically. This may, however, be tedious because of the many terms of the function. Instead, because of the above theorem, we know that we can successfully apply the bisection method to Equation 15 to find a root of this function. The authors have tested this successfully, even with data sets with hundreds of thousands of variables. Also, we may use either root, if there are more than one, since they all give $\text{SNR}(a) = \sigma$, although we may want to find the one that minimizes $|a - 1|$.

Thus, we would encapsulate Algorithm 1 in a bisection loop in order to control the signal-to-noise ratio.

5.4. Correlation

We control the correlation structure of X_0 by, e.g., letting $X_0 \sim \mathcal{N}(\mu, \Sigma)$. Since we let $X_j = \omega_j X_{0,j}$ for all $1 \leq i \leq p$, it follows that $\text{cor}(X_l, X_m) = \text{cor}(\omega_l X_{0,l}, \omega_m X_{0,m})$.

6. The package pylearn-simulate

The package **pylearn-simulate** contains five main modules. This section describes the five modules, and how they implement the theory described above.

For the sake of clarity, the modules are presented in alphabetical order. However, we advise a reader interested in directly using our package to jump directly to Section 6.4, presenting **pylearn-simulate**'s main module, or to Section 7 in which three examples are detailed.

6.1. `beta.py`

The first module allows the generation of random (weight) vectors. It can, e.g., be used to

generate the true minimizer, β^*, mentioned in Section 1.

Its main function is defined as:

```
def random(shape, density = 1.0, rng = utils.RandomUniform(0, 1),
           sort = False, normalise = False):
```

where `shape` is the shape of the underlying data, e.g., p-by-1 would be `shape = (p, 1)`; `density` is a value between 0 and 1 that determines the fraction of non-zero elements of the returned vector (default is `density = 1.0`, a completely dense vector; `density = 0.0` would be a vector of zeros); `rng` determines the random number generator to use (`rng` is a function or callable that takes a number of integers as input, the shape of the array of random numbers to return); `sort` is a Boolean that determines whether or not to sort the output vector (will sort each axis in turn in ascending order); `normalise` is a Boolean that determines whether or not to normalize the output vector. If `normalise = True`, the output vector will have unit ℓ_2-norm.

6.2. `correlation_matrices.py`

The second module is used to generate random correlation matrices with a particular structure. When applied to the generation of a correlation matrix Σ, the initial matrix, X_0, can be sampled from $\mathcal{N}(\mu, \Sigma)$, for some mean vector μ.

This module contains two ways to generate correlation matrices, described by Hardin, Ramon Garcia, and Golan (2013). The two ways to generate correlation matrices are:

1. A correlation matrix with a *constant correlation structure*, i.e., such that

$$\Sigma_k = \begin{bmatrix} 1 & \rho_k & \dots & \rho_k \\ \rho_k & 1 & \dots & \rho_k \\ \vdots & \vdots & \ddots & \vdots \\ \rho_k & \rho_k & \dots & 1, \end{bmatrix},$$

   where Σ_k is a block diagonal element of Σ, and ρ_k is the average correlation.

2. A correlation matrix with a *Toeplitz correlation structure*, i.e., such that

$$\Sigma_k = \begin{bmatrix} 1 & \rho_k & \rho_k^2 & \rho_k^3 & \dots & \rho_k^{p_k-1} \\ \rho_k & 1 & \rho_k & \rho_k^2 & \dots & \rho_k^{p_k-2} \\ \rho_k^2 & \rho_k & 1 & \rho_k & \dots & \rho_k^{p_k-3} \\ \rho_k^3 & \rho_k^2 & \rho_k & 1 & \dots & \rho_k^{p_k-4} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \rho_k^{p_k-1} & \rho_k^{p_k-2} & \rho_k^{p_k-3} & \rho_k^{p_k-4} & \dots & 1, \end{bmatrix},$$

   where Σ_k is a block diagonal element of Σ, and ρ_k is the average correlation between adjacent variables. The correlation thus decreases exponentially as a function of the *distance* between the variables.

Background noise and off-block-diagonal noise may also be added to the correlation matrices. The k stands for the kth group of variables; the module indeed allows the user to generate

correlation matrices with blocks of variables, each one of them having a different correlation structure. See Hardin *et al.* (2013) for details.

This module contains the following functions:

```
def constant_correlation(p = [100], rho = [0.05], delta = 0.05, eps = 0.5,
                         random_state = None):
```

where `p` is a list or tuple with the number of variables for each group; `rho` is a list or tuple with the average correlation between variables in a group such that `rho[k]`$\in [0, 1)$; `delta` $\in$ [0, `min(rho)`) defines the baseline noise between groups; `eps` $\in [0, 1-$`max(rho)`$)$ is the entry-wise random noise with mean `delta` and variance `eps`$^2/10$ (where 10 is the dimension of the noise space, selected arbitrarily); and `random_state` is an instance of `numpy.random.RandomState` (if omitted, or `None`, the default NumPy random number generator will be used instead).

```
def toeplitz_correlation(p = [100], rho = [0.05], eps = 0.5,
                         random_state = None):
```

where `p` is a list or tuple with the number of variables for each group; `rho` is a list or tuple with the average correlation between variables in a group such that `rho[k]`$\in [0, 1)$; `eps` $\in$ (0, (1$-$`max(rho)`)/(1+`max(rho)`) defines the maximum entry-wise random noise. The noise is approximately normally distributed with zero mean and variance `eps`$^2/10$; and `random_state` is an instance of `numpy.random.RandomState` (if omitted, or `None`, the default NumPy random number generator will be used instead).

6.3. `functions.py`

This module defines a set of penalties that can be combined with (added to) the loss function. The functions are implemented as classes and they all inherit from `Function`, which requires them to implement a function `grad` that returns the gradient (or a subgradient).

The different penalties currently implemented are detailed below.

The class

```
class L1(Function):
    def __init__(self, l, rng = utils.RandomUniform(-1, 1))
```

represents

$$\mathtt{l}\|\beta\|_1,$$

the ℓ_1-norm penalty, where `l` is the regularization constant (denoted λ_{ℓ_1} above); `rng` is a random number generator to use when computing a subgradient. `rng` is a function or callable that takes a number of integers as input, the shape of the array of random numbers to return. Default `rng` is random uniform numbers between -1 and 1.

The class

```
class SmoothedL1(Function):
    def __init__(self, l, mu = utils.TOLERANCE):
```

represents

$$\mathtt{l}\left(\langle\beta,\alpha^*\rangle - \frac{\mathtt{mu}}{2}\|\alpha^*\|_2^2\right),$$

the Nesterov-smoothed ℓ_1 penalty, where `l` is the regularization constant and `mu` is the regularization constant for the Nesterov smoothing.

The class

```
class L2(Function):
    def __init__(self, l, rng = utils.RandomUniform(0, 1)):
```

represents

$$\mathtt{l}\|\beta\|_2,$$

the ℓ_2-norm penalty, where `l` is the regularization constant; `rng` is a random number generator that is used when a subgradient is computed. `rng` is a callable that takes a number of integers as input, the shape of the array of random numbers to return. Default `rng` is random uniform numbers between 0 and 1.

The class

```
class L2Squared(Function):
    def __init__(self, l):
```

represents

$$\frac{\mathtt{l}}{2}\|\beta\|_2^2,$$

the squared ℓ_2-norm penalty, where `l` is the regularization constant (denoted λ_{ℓ_2} above).

The class

```
class TotalVariation(Function):
    def __init__(self, l, A, rng = utils.RandomUniform(0, 1), **kwargs):
```

represents

$$\mathtt{l}\sum_{j=1}^{p}\|\nabla\beta_j\|_2,$$

the total variation penalty, where `l` is the regularization constant (denoted λ_π above); `A` is the linear operator for the total variation penalty, as specified in Section B.1, and obtained from the static methods `TotalVariation.A_from_shape` or `TotalVariation.A_from_subset_mask`; `rng` is a random number generator that is used when a subgradient is computed. `rng` is a callable that takes a number of integers as input, the shape of the array of random numbers to return. Default `rng` is random uniform numbers between 0 and 1.

The class

```
class GroupLasso(Function):
    def __init__(self, l, A, rng = utils.RandomUniform(-1, 1), **kwargs):
```

represents

$$\texttt{l}\sum_{g=1}^{G}\texttt{weights[g]}\|\beta_g\|_2,$$

the overlapping group LASSO, where `l` is the regularization constant (denoted λ_π above); `A` is the linear operator for the overlapping group LASSO penalty, as specified in Section B.2, and obtained from `GroupLasso.A_from_groups`; `rng` is a random number generator that is used when a subgradient is computed. `rng` is a function or callable that takes a number of integers as input, the shape of the array of random numbers to return. Default `rng` is random uniform numbers between -1 and 1. `weights` is a list or tuple provided to `GroupLasso.A_from_groups` that gives a weight to each group; and β_g is a vector with the variables of group g.

The class

```
class SmoothedTotalVariation(TotalVariation, NesterovFunction):
    def __init__(self, l, A, mu = utils.TOLERANCE):
```

represents

$$\texttt{l}\left(\langle A\beta, \alpha^*\rangle - \frac{\texttt{mu}}{2}\|\alpha^*\|_2^2\right),$$

the Nesterov-smoothed total variation penalty, where `l` is the regularization constant (denoted λ_π above); `A` is the linear operator for the total variation penalty, as specified in Section B.1, and obtained from `TotalVariation.A_from_shape` or `TotalVariation.A_from_subset_mask`, two static methods; `mu` is the regularization constant for the Nesterov smoothing.

Note that `SmoothedTotalVariation` inherits from `NesterovFunction`, a base class for Nesterov-smoothed complex penalties.

The class

```
class SmoothedGroupLasso(GroupLasso, NesterovFunction):
    def __init__(self, l, A, mu = utils.TOLERANCE):
```

represents

$$\texttt{l}\left(\langle A\beta, \alpha^*\rangle - \frac{\texttt{mu}}{2}\|\alpha^*\|_2^2\right),$$

the Nesterov-smoothed overlapping group LASSO penalty, where `l` is the regularization constant (denoted λ_π above); `A` is the linear operator for the overlapping group LASSO penalty, as specified in Section B.2, and obtained from `GroupLasso.A_from_groups`.

6.4. `simulate.py`

This is the main module of **pylearn-simulate**, and the starting point when generating simulated data. It contains a class `LinearRegressionData` that generates simulated data with a linear regression (linear least squares) objective function, and accepts any combination of functions (from `functions.py`) to penalize the loss.

`LinearRegressionData` is defined as

```
class LinearRegressionData(SimulatedData):
    def __init__(self, penalties, X0, e, snr = None, intercept = False):
```

where `penalties` is a list or tuple with the penalties; `X0` is the initial candidate data set, as explained in Section 1; `e` $= X\beta - y$ is the residual vector, as explained in Section 4.1; `snr` is the desired signal-to-noise ratio; `intercept` is a Boolean that determines whether or not to account for an intercept. If `intercept = True`, then the first column of `X0` must contain only ones.

Once a `LinearRegressionData` object is constructed, the data is generated by calling

```
>>> lrd = LinearRegressionData(...)
>>> X, y, beta_star, e = lrd.load(beta0)
```

where `beta_star` is the true, known regression vector β^*.

6.5. `utils.py`

This module contains some helper functions, such as the random number generators. It contains the following classes and functions:

```
class RandomUniform(RandomNumberGenerator):
    def __init__(self, a = 0, b = 1, random_state = None):
```

where `a` and `b` are the limits within which the random uniform numbers are sampled, and `random_state` is an optional `numpy.random.RandomState` object to use when sampling points (this is useful for setting a seed, for instance).

```
class ConstantValue(RandomNumberGenerator):
    def __init__(self, val, random_state = None):
```

where `val` is a single value that is returned for every call to this number generator. The value of `random_state` is never used here.

```
def find_bisect_interval(f, low = -1.0, high = 1.0, maxiter = 100):
```

find values of `low` and `high` such that `sign(f(low)) != sign(f(high)`.

7. Examples

The main benefit of generating data with **pylearn-simulate** is that we know everything about the generated data. In particular, we know the true minimizer, β^*, and we know the Lagrange multipliers, or regularization parameters, λ_{ℓ_1}, λ_{ℓ_2} and λ_π. There is no need to use, e.g., cross-validation to find the parameters, and we know directly if the $\beta^{(k)}$ that our minimizing algorithm finds is close to the true β^* or not.

We illustrate this main point by three simulations, in which we also demonstrate how to use **pylearn-simulate**. Note that due to differences in core packages such as **NumPy** or **SciPy** slightly different random numbers and corresponding plots may be obtained. An example in R is available in the supplementary material and in the GitHub repository (Löfstedt *et al.* 2014a).

7.1. Comparison to the classical approach

Traditionally, simulated data for linear regression problems is generated as follows: The independent data matrix, X, is sampled from some multivariate distribution, a regression vector β^* is generated either with or without sparsity, structure, etc. The noise vector, ε, is usually Gaussian, $\varepsilon \sim \mathcal{N}(0, \sigma^2 I)$, for some value of σ. Finally, the response variable, y, is computed as

$$y = X\beta^* + \varepsilon. \tag{20}$$

The problem then is to find a β such that a certain penalized linear regression problem is minimized.

The main issue is that the generated or sampled data, X, will not adhere to any sparsity or structure constraints that we may have on the solution β^*. In essence, the data generated does not fit the mathematical model when penalties are involved. A computational problem also arises, because the regularization parameters for the penalties are not known in advance, and finding the minimum is thus going to be computationally expensive, since the parameters will have to be found by, e.g., cross-validation.

We will in this example simulate data by using **pylearn-simulate** for linear regression with an elastic net penalty. We will then compare the solution where the regularization constants are found by grid search and cross-validation to that of the theoretical solution.

Import packages:

```
>>> import simulate
>>> import numpy as np
>>> np.random.seed(42)
>>> rs = np.random.RandomState(42)
>>> rng01 = simulate.utils.RandomUniform(0, 1, random_state = rs)
>>> rng_11 = simulate.utils.RandomUniform(-1, 1, random_state = rs)
```

Generate a start vector, β_0, a candidate data set, X_0, and the residual vector, ε:

```
>>> n, p = 50, 100
>>> beta = simulate.beta.random((p + 1, 1), density = 0.5, sort = True,
...   rng = rng01)
>>> Sigma = simulate.correlation_matrices.constant_correlation(p = p,
...   rho = 0.1, eps = 0.01, random_state = rs)
>>> X0 = rs.multivariate_normal(np.zeros(p), Sigma, n)
>>> X0 = np.hstack((np.ones((n, 1)), X0))
>>> e = 0.1 * rs.randn(n, 1)
```

A column of ones is added to `X0` in order to deal with the intercept.

Create the penalties:

```
>>> lambda_l1 = 0.618
>>> l1 = simulate.functions.L1(lambda_l1, rng = rng_11)
>>> l2 = simulate.functions.L2Squared(1.0 - lambda_l1)
```

Create the loss function:

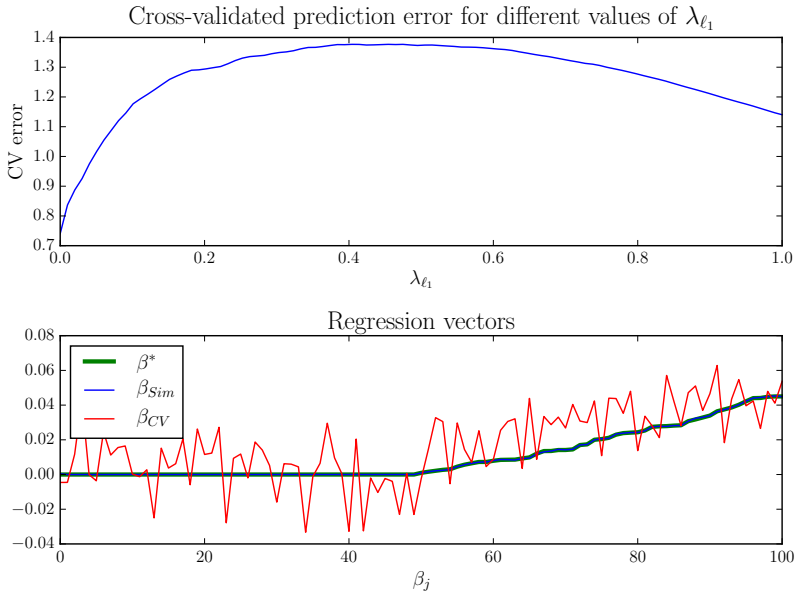

Figure 1: Upper panel: The cross-validated prediction error (sum-of-squares error) for different values of λ_{ℓ_1}. Lower panel: The regression vectors for the true β^* (green), the regression vector found when using the true regularization parameters, β_{Sim}, (blue) and the regression vector found when using the λ_{ℓ_1} that gave the smallest cross-validated prediction error, β_{CV} (red).

```
>>> lr = simulate.LinearRegressionData([l1, l2], X0, e, snr = 5.0,
...   intercept = True)
```

Finally, generate the data:

```
>>> X, y, beta_star, e = lr.load(beta)
```

We ran a grid search with 100 values of `lambda_l1` in the interval $[0, 1]$ and for every value of `lambda_l1` we performed a 10-fold cross-validation. Any minimization algorithm can be used here, and we used the fast iterative shrinkage-thresholding algorithm (FISTA) from **pylearn-parsimony** (Löfstedt *et al.* 2014b), an optimization library for machine learning with structured and sparsity-inducing penalties.

The prediction error is plotted against the different values of the regularization parameter in Figure 1 (upper panel). The minimum cross-validated prediction error was found when `lambda_l1` $= 0.0$, i.e., no ℓ_1 penalization, and a strong ℓ_2 penalization. The cross-validated prediction error was

$$\frac{1}{K} \sum_{k=1}^{K} \|y_k - \widehat{y}_k\|_2^2 \approx 0.74,$$

where $\widehat{y}_k$ was the predicted y from the left out cross-validation group. There were $K = 10$ cross-validation groups.

The regression vector that corresponds to the smallest cross-validated prediction error of y, denoted β_{CV}, is presented in Figure 1 (lower panel) together with β^*, the true regression

vector, and the regression vector found when using the true regularization parameters, denoted β_{Sim}. Note how different the regression vector found by cross-validation is compared to the true β^*; note also that β_{Sim} is indistinguishable from β^*.

The Python code for this example is contained in the supplementary material, in the file `comparison_to_classical_approach.py`.

In conclusion, the proposed way of simulating data, strictly adhering to a given optimization problem, presents advantages that are highly valuable when conducting a simulated experiment. First, since the values of the regularization parameters are known and set by the users, they can be given to an optimization algorithm and the result be compared to the known minimizer of the function. Apart from the practicality of doing so, large amounts of time will be saved by avoiding techniques such as cross-validation to select optimal values of the regularization parameters. Second, one can test in a controlled manner the robustness, or sensitivity of an optimization method to the "wrong" values of the regularization parameters, since one would know exactly how far the regularization parameter values are from the true values. Third, knowing exactly what is minimized (i.e., the exact expression of the function) and its minimizer allows us to know exactly how a minimization method fares: The experimenter knows at each iteration how far an algorithm is from the minimum of the function and, at the end of the optimization process, he/she will be able to derive the exact speed at which the optimum was found.

These advantages could benefit many current high level scientific projects. E.g., the one by Dohmatob, Gramfort, Thirion, and Varoquaux (2014), in which the authors compared the performance of several optimization algorithms for solving a logistic regression problem with TV and ℓ_1 penalties. It is widely known that this problem has no explicit solution. This is a typical case in which the proposed approach would have saved time and improved the quality of the comparison. Indeed, the authors studied the convergence of the algorithms for parameters close to the optimal parameters, set by 10-fold cross-validation. The proposed approach would have allowed them to directly plug the optimal parameters in the optimization algorithms and/or test the robustness of the chosen optimization algorithms to values of the parameters that would have been chosen to differ from the optimal values.

7.2. Elastic net and total variation

In this example we simulate data to fit the function

$$f(\beta) = \frac{1}{2}\|X\beta - y\|_2^2 + (1 - \lambda_{\ell_2})\|\beta\|_1 + \frac{\lambda_{\ell_2}}{2}\|\beta\|_2^2 + \lambda_{\text{TV}}\,\text{TV}(\beta), \tag{21}$$

in which $\lambda_{\ell_2} = 0.5$ and $\lambda_{\text{TV}} = 1.0$. This is thus elastic net with a TV penalty. We let the ℓ_1 parameter be $1 - \lambda_{\ell_2}$. We will vary the values of the regularization parameters in an interval around their "true" values and compute $f(\beta^{(k)}) - f(\beta^*)$ for each of these values.

We use **pylearn-simulate** to generate data that fit this loss function as follows.

Import the required packages:

```
>>> import simulate
>>> import numpy as np
>>> np.random.seed(42)
>>> rs = np.random.RandomState(42)
```

```
>>> rng01 = simulate.utils.RandomUniform(0, 1, random_state = rs)
>>> rng_11 = simulate.utils.RandomUniform(-1, 1, random_state = rs)
```

Generate a start vector, β_0, a candidate data set, X_0, and the residual vector, ε:

```
>>> shape = (4, 4, 4)
>>> n, p = 48, np.prod(shape)
>>> beta = simulate.beta.random((p, 1), density = 0.5, sort = True,
...   rng = rng01)
>>> Sigma = simulate.correlation_matrices.constant_correlation(p = p,
...   rho = 0.01, eps = 0.001, random_state = rs)
>>> X0 = rs.multivariate_normal(np.zeros(p), Sigma, n)
>>> e = rs.randn(n, 1)
```

Generate the linear operator for total variation:

```
>>> A = simulate.functions.TotalVariation.A_from_shape(shape)
```

Create the penalties:

```
>>> lambda_l2 = 0.5
>>> lambda_l1 = 1.0 - lambda_l2
>>> lambda_tv = 1.0
>>> l1 = simulate.functions.L1(lambda_l1, rng = rng_11)
>>> l2 = simulate.functions.L2Squared(lambda_l2)
>>> tv = simulate.functions.TotalVariation(lambda_tv, A, rng = rng01)
```

Create the loss function:

```
>>> lr = simulate.LinearRegressionData([l1, l2, tv], X0, e, snr = 3.0,
...   intercept = False)
```

Finally, generate the data:

```
>>> X, y, beta_star, e = lr.load(beta)
```

We minimize the function in Equation 21 on the resulting data by using CONESTA (Hadj-Selem *et al.* 2018) (which, roughly, is FISTA with continuation of the smoothing parameter μ) from **pylearn-parsimony** (Löfstedt *et al.* 2014b), with the smallest value of the smoothing parameter set to $\mu = 5 \cdot 10^{-8}$.

We minimize Equation 21 for 21 different values of each of the regularization parameters. We vary the parameters over `lambda_l2` $\in [\lambda_{\ell_2} - 0.25, \lambda_{\ell_2} + 0.25]$ and `lambda_tv` $\in [\lambda_{\text{TV}} - 0.25, \lambda_{\text{TV}} + 0.25]$.

The result is shown in Figure 2, and we see that the solution that gives the smallest function value is at precisely the true values of $\lambda_{\ell_2} a$ and λ_{TV}.

The complete Python code for this example is found in the supplementary material, in the file `linear_regression_elastic_net_total_variation.py`.

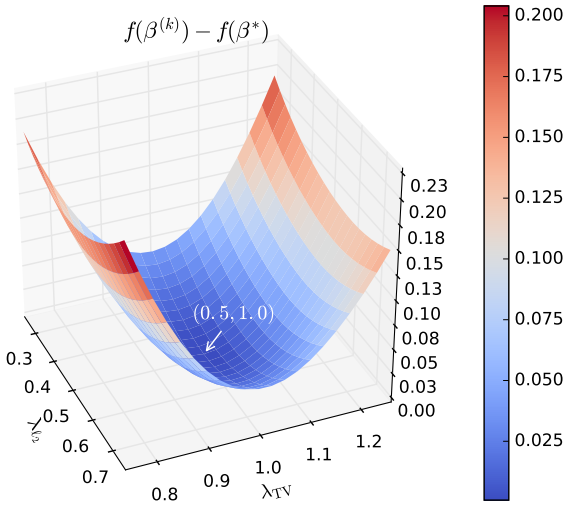

Figure 2: An illustration of the benefit of using simulated data as described in this article. The minimum solution to Equation 21 is found when using the regularization parameters used in the construction of the simulated data. These 48×64 data had no correlation between variables and the following characteristics: 50 % sparsity, signal-to-noise ratio 3, $\lambda_{\ell_2} = 0.5$, and $\lambda_{\text{TV}} = 1.0$.

7.3. Elastic net and smoothed group LASSO

In the third example we simulate data for the function

$$f(\beta) = \frac{1}{2}\|X\beta - y\|_2^2 + \lambda_{\ell_1}\|\beta\|_1 + \frac{1 - \lambda_{\ell_1}}{2}\|\beta\|_2^2 + \lambda_{\text{GL}}\,\text{GL}_\mu(\beta), \tag{22}$$

where GL_μ is the smoothed (overlapping) group LASSO function; and in which $\lambda_{\ell_1} = 0.618$ and $\lambda_{\text{GL}} = 1.618$ (in this example we let the ℓ_2 parameter be $1 - \lambda_{\ell_1}$). We will vary the values of the regularization parameters in an interval around their "true" values and compute $f(\beta^{(k)}) - f(\beta^*)$ for each of these values.

We use **pylearn-simulate** to generate data that fit this loss function as follows.

Import the required packages:

```
>>> import simulate
>>> import numpy as np
>>> np.random.seed(42)
>>> rs = np.random.RandomState(42)
>>> rng01 = simulate.utils.RandomUniform(0, 1, random_state = random_state)
>>> rng_11 = simulate.utils.RandomUniform(-1, 1, random_state = random_state)
```

Generate a start vector, β_0, a candidate data set, X_0, the residual vector, ε, and required parameters. The first step is to define the size of the data:

```
>>> n, p = 48, 64 + 1
```

and the groups

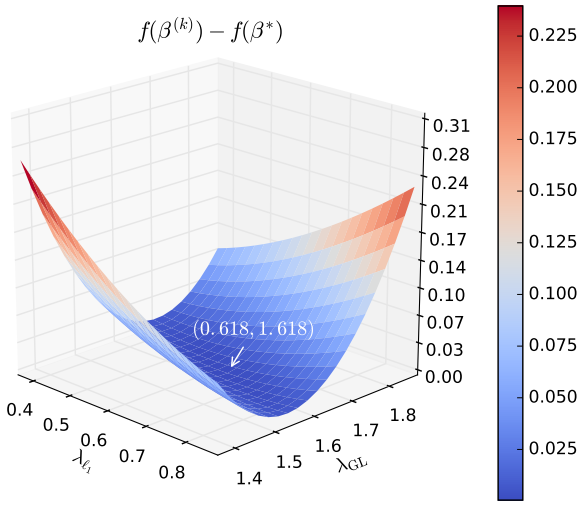

Figure 3: Another illustration of the benefit of using simulated data as described in this article. The minimum solution to Equation 22 is found when using the regularization parameters used in the construction of the simulated data. These 48×65 data had an intercept variable, no correlation between variables and the following characteristics: 50 % sparsity, signal-to-noise ratio 2, $\lambda_{\ell_1} = 0.618$, and $\lambda_{\mathrm{GL}} = 1.618$.

```
>>> groups = [range(1, 2 * p / 3), range(p / 3, p)]
```

After that, we generate the data set with a column of ones in `X0` to take the intercept into account:

```
>>> beta = simulate.beta.random((p - 1, 1), density = 0.5, sort = True,
...   rng = rng01)
>>> beta = np.vstack((rs.rand(1, 1), beta))
>>> Sigma = simulate.correlation_matrices.constant_correlation(p = p - 1,
...   rho = 0.01, eps = 0.001, random_state = rs)
>>> X0 = rs.multivariate_normal(np.zeros(p - 1), Sigma, n)
>>> X0 = np.hstack((np.ones((n, 1)), X0))
>>> e = rs.randn(n, 1)
```

Generate the linear operator for overlapping group LASSO:

```
>>> A = simulate.functions.SmoothedGroupLasso.A_from_groups(p, groups,
...   weights = None, penalty_start = 1)
```

Create the penalties, where `lambda_l1`, `lambda_l2` and `lambda_gl` are the parameters used for the ℓ_1, the ℓ_2, and the group lasso penalties, respectively:

```
>>> lambda_l1 = 0.618
>>> lambda_l2 = 1.0 - lambda_l1
>>> lambda_gl = 1.618
>>> l1 = simulate.functions.L1(lambda_l1, rng = rng_11)
```

```
>>> l2 = simulate.functions.L2Squared(lambda_l2)
>>> gl = simulate.functions.SmoothedGroupLasso(lambda_gl, A,
...   mu = simulate.utils.TOLERANCE)
```

Create the loss function:

```
>>> lr = simulate.LinearRegressionData([l1, l2, gl], X0, e, snr = 2.0,
...   intercept = True)
```

Generate simulated data:

```
>>> np.random.seed(42)
>>> X, y, beta_star, e = lr.load(beta)
```

We again minimize the function in Equation 22 on these data by using CONESTA from **pylearn-parsimony** (Löfstedt *et al.* 2014b), with the smallest, and true value of the smoothing parameter set to $\mu = 5 \cdot 10^{-8}$. See the `examples` directory in **pylearn-simulate** for more details.

The result is shown in Figure 3, and we again see that the solution that gives the smallest function value is at precisely the true values of λ_{ℓ_1} and λ_{GL}.

The complete Python code for this example is found in the supplementary material, in the file `linear_regression_elastic_net_group_lasso.py`.

8. Discussion and conclusions

The technique that we have used in order to generate the simulated data in this paper is very useful when testing new optimization methods. Indeed, it allows us to clearly measure the performance of the optimization method while avoiding an arbitrary choice of the penalization constants. This means the choice of penalization will not impact the quality of the solution. In fact, this completely avoids the use of cross-validation, or other resampling techniques, something that may be computationally expensive, and that does not guarantee optimality.

Additionally, if the candidate matrix, X_0, is a real data set, it is possible to use the proposed technique to generate a quasi-real data set and thus under quasi-real conditions control all the parameters of the problem. In particular, since we can generate different choices of β^*, we can test different structured and sparse penalties and see how they perform on the particular data.

Also, since we know the problem exactly, we are able to numerically study the behavior of cross-validation, or other resampling techniques, on our data. This may help us understand how cross-validation performs on different types of data under different model hypotheses.

The presented Python package, **pylearn-simulate**, offers a simple object oriented interface with which it is possible to generate data that fits any combination of penalties with a linear least-squares loss function. It is easy to extend the package and add new penalties, thanks to the object oriented interface. It is also possible to extend the package, as discussed below, and to add other loss functions.

The aim of this paper was to discuss how to generate simulated data for optimization problems with the linear least-squares loss with ℓ_1, ℓ_2, and a complex penalty such as e.g. TV. We note that Theorem 5.1 only covers these penalties, but also that the formulation of the complex

penalties is very general, and therefore covers a large class of penalties. Theorem 5.1 can likely be generalized to other losses and penalties as well, but that was out of the scope of this paper.

Finally, we note that other machine learning methods, such as, e.g., logistic regression, are different from the one with linear regression presented here. In logistic regression, the y vector is implicitly related to the X matrix. It is possible to find an exact solution for the minimization problem, but without any control of the correlation structure or on the signal-to-noise ratio. Therefore, a different approach is needed in this context and for other machine learning methods as well. This is something that will occupy the authors' attention in future research.

Acknowledgments

This work was supported by grants from the French National Research Agency: ANR GENIM (ANR-10-BLAN-0128), ANR IA BRAINOMICS (ANR-10-BINF-04), and a European Commission grant: MESCOG (FP6 ERA-NET NEURON 01 EW1207).

The authors Tommy Löfstedt, Vincent Guillemot, and Fouad Hadj-Selem contributed equally to this publication.

References

Bach F, Jenatton R, Mairal J, Obozinski G (2011). "Convex Optimization with Sparsity-Inducing Norms." In S Sra, S Nowozin, SJ Wright (eds.), *Optimization for Machine Learning*. MIT Press. URL http://www.di.ens.fr/~fbach/opt_book.pdf.

Chen X, Liu H (2011). "An Efficient Optimization Algorithm for Structured Sparse CCA, with Applications to eQTL Mapping." *Statistics in Biosciences*, **4**(1), 3–26. doi:10.1007/s12561-011-9048-z.

Dohmatob E, Gramfort A, Thirion B, Varoquaux G (2014). "Benchmarking Solvers for TV-L1 Least-Squares and Logistic Regression in Brain Imaging." In *Pattern Recoginition in Neuroimaging*. IEEE, Tübingen, Germany.

Hadj-Selem F, Löfstedt T, Dohmatob E, Frouin V, Dubois M, Guillemot V, Duchesnay E (2018). "Continuation of Nesterov's Smoothing for Regression with Structured Sparsity in High-Dimensional Neuroimaging." *IEEE Transactions on Medical Imaging*. doi:10.1109/TMI.2018.2829802. Forthcoming.

Hardin J, Ramon Garcia S, Golan D (2013). "A Method for Generating Realistic Correlation Matrices." *The Annals of Applied Statistics*, **7**(3), 1733–1762. doi:10.1214/13-aoas638.

Hiriart-Urrut JB, Lemaréchal C (2004). *Fundamentals of Convex Analysis*. 2nd edition. Springer-Verlag.

Jones E, Oliphant T, Peterson P (2001). "**SciPy**: Open Source Scientific Tools for Python." URL http://www.scipy.org/.

Löfstedt T, Guillemot V, Frouin V, Duchesnay E, Hadj-Selem F (2014a). "**pylearn-Simulate**." GitHub repository, https://github.com/neurospin/pylearn-simulate.

Löfstedt T, Guillemot V, Hadj-Selem F, Li J, Frouin V, Duchesnay E (2014b). "**pylearn-Parsimony**." GitHub repository, https://github.com/neurospin/pylearn-parsimony.

Nesterov Y (2004). "Smooth Minimization of Non-Smooth Functions." *Mathematical Programming*, **103**(1), 127–152. doi:10.1007/s10107-004-0552-5.

Nesterov Y (2013). "Gradient Methods for Minimizing Composite Functions." *Mathematical Programming*, **140**(1), 125–161. doi:10.1007/s10107-012-0629-5.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

RStudio (2013). "**RStudio**: Integrated Development for R." **RStudio**, Inc., URL https://www.RStudio.com/.

Rudin L, Osher S, Fatemi E, Monica S (1992). "Nonlinear Total Variation Based Noise Removal Algorithms." *Physica D: Nonlinear Phenomena*, **60**(1–4), 259–268. doi:10.1016/0167-2789(92)90242-f.

van der Walt S, Colbert SC, Varoquaux G (2011). "The **NumPy** Array: A Structure for Efficient Numerical Computation." *Computing in Science & Engineering*, **13**(2), 22–30. doi:10.1109/mcse.2011.37.

van Rossum G, *et al.* (2011). *Python Programming Language*. URL https://www.python.org/.

Yuan M, Lin Y (2006). "Model Selection and Estimation in Regression with Grouped Variables." *Journal of the Royal Statistical Society B*, **68**(1), 49–67. doi:10.1111/j.1467-9868.2005.00532.x.

A. LASSO

The function

$$f(\beta) = \frac{1}{2}\|X\beta - y\|_2^2 + \lambda_{\ell_1}\|\beta\|_1$$

is known as the LASSO problem.

Nesterov (2013) addressed how to simulate data for this case, and we will therefore not go into details. Instead we will simply adapt it to our notation and explain some steps that are not obvious.

The principle behind Nesterov's idea is as follows: First, define the error to be $\varepsilon = X\beta - y$, in the model between $X\beta$ and y, such that it is independent from β^*. Then, select acceptable values for the columns of X such that zero belongs to the subdifferential of f at β^*, with subdifferential

$$\begin{aligned}\partial f(\beta) &= X^\top(X\beta - y) + \lambda_{\ell_1}\partial\|\beta\|_1 \\ &= X^\top\varepsilon + \lambda_{\ell_1}\partial\|\beta\|_1.\end{aligned}$$

At β^* we have such that

$$0 \in X^\top\varepsilon + \lambda_{\ell_1}\partial\|\beta^*\|_1, \tag{23}$$

and we stress again that $X^\top\varepsilon$ does not depend on β^*. We distinguish two cases:

First case: We consider a variable $\beta_j^* \neq 0$, the jth element of β^*. With $\beta_j^* \neq 0$ it follows that $\partial|\beta_j^*| = \{\text{sign}(\beta_j^*)\}$, and thus that we can write

$$0 = X_j^\top\varepsilon + \lambda_{\ell_1}\,\text{sign}(\beta_j^*),$$

because of Equation 23, with X_j the jth column of X.

Second case: We consider the case when $\beta_j^* = 0$. We note that the subdifferential of $|\beta_j^*|$ when $\beta_j^* = 0$ is

$$\partial|\beta_j^*| = [-1, 1],$$

and thus from Equation 23 we see that

$$0 \in X_j^\top\varepsilon + \lambda_{\ell_1}[-1, 1]. \tag{24}$$

Solution

The candidate matrix X_0 will serve as a first unscaled version of X and we have such that $X_j = \omega_j X_{0,j}$, for all $1 \leq j \leq p$.

If $\beta_j^* \neq 0$, then $X_j^\top\varepsilon + \lambda_{\ell_1}\,\text{sign}(\beta_j^*) = 0$ and thus

$$X_j^\top\varepsilon = -\lambda_{\ell_1}\,\text{sign}(\beta_j^*)$$

and since $X_j = \omega_j X_{0,j}$ we have

$$\omega_j = \frac{-\lambda_{\ell_1}\,\text{sign}(\beta_j^*)}{X_{0,j}^\top\varepsilon}.$$

If $\beta_j^* = 0$, we use Equation 24 and have

$$X_j^\top \varepsilon \in \lambda_{\ell_1}[-1, 1].$$

Thus, with $X_j = \omega_j X_{0,j}$ we obtain

$$\omega_j X_{0,j}^\top \varepsilon \in \lambda_{\ell_1}[-1, 1],$$

or equivalently

$$\omega_j \sim \frac{\lambda_{\ell_1} \mathcal{U}(-1, 1)}{X_{0,j}^\top \varepsilon}.$$

Once X is generated, we let $y = X\beta^* - \varepsilon$.

B. Two complex penalties

B.1. Total variation

Gradient of non-smooth total variation

The TV penalty, for a discrete β, is defined as

$$\mathrm{TV}(\beta) = \sum_{j=1}^{p} \| \operatorname{grad}(\beta_j) \|_2, \tag{25}$$

where $\operatorname{grad}(\beta_j)$ is the discrete spatial gradient at point β_j. It is usually expressed as a forward difference discrete gradient, i.e., such that

$$\begin{aligned} \mathrm{TV}(\beta) &= \sum_{j=1}^{p} \| \operatorname{grad}(\beta_j) \|_2 \\ &= \sum_{j_1=1}^{p_1-1} \cdots \sum_{j_D=1}^{p_D-1} \sqrt{(\beta_{j_1+1,\ldots,j_D} - \beta_{j_1,\ldots,j_D})^2 + \cdots + (\beta_{j_1,\ldots,j_D+1} - \beta_{j_1,\ldots,j_D})^2}, \end{aligned}$$

where p_d is the number of variables in the dth dimension, for $d = 1, \ldots, D$, with D dimensions. We will first illustrate this in the 1-dimensional case. In this case $\|x\|_2 = \sqrt{x^2} = |x|$, since $x \in \mathbb{R}$. We thus have

$$\mathrm{TV}(\beta) = \sum_{j=1}^{p-1} |\beta_{j+1} - \beta_j|,$$

We note that if we define A, a $(p-1) \times p$ matrix, as

$$A = \begin{bmatrix} -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & 1 \end{bmatrix},$$

1st layer:

1	2	3	4
5	6	7	8
9	10	11	12

2nd layer:

13	14	15	16
17	18	19	20
21	22	23	24

Figure 4: The 24-dimensional regression vector β representing a $2 \times 3 \times 4$ image.

then

$$\mathrm{TV}(\beta) = \sum_{j=1}^{p-1} |\beta_{j+1} - \beta_j| = \sum_{g=1}^{G} \|A_g \beta\|_2,$$

where $G = p - 1$ and A_g is the gth row of A.

Thus, we use Theorem 4.3 and obtain

$$\partial TV(\beta) = A^\top \begin{bmatrix} \partial \|A_1\beta\|_2 \\ \vdots \\ \partial \|A_G\beta\|_2 \end{bmatrix} = A^\top \begin{bmatrix} \partial \, |\beta_2 - \beta_1| \\ \vdots \\ \partial \, |\beta_p - \beta_{p-1}| \end{bmatrix},$$

in which we recall Equation 6 and obtain that

$$\partial |x| = \begin{cases} \{\mathrm{sign}(x)\} & \text{if } |x| > 0, \\ [-1, 1] & \text{if } |x| = 0. \end{cases}$$

The general case will be illustrated with a small example using a 3-dimensional image. A 24-dimensional regression vector β is generated, that represents a $2 \times 3 \times 4$ *image*. The image, with linear indices indicated, is given in Figure 4.

We note, when using the linear indices, that β_1 and β_2 are neighbors in the 1st dimension, that β_1 and β_5 are neighbors in the 2nd dimension and that β_1 and β_{13} are neighbors in the 3rd dimension. Using 3-dimensional indices (i.e., such as $\beta_{i,j,k}$) the penalty becomes

$$TV(\beta) = \sum_{j_1=1}^{p_1-1} \sum_{j_2=1}^{p_2-1} \sum_{j_3=1}^{p_3-1} \sqrt{\begin{array}{c} (\beta_{j_1+1,j_2,j_3} - \beta_{j_1,j_2,j_3})^2 \\ +(\beta_{j_1,j_2+1,j_3} - \beta_{j_1,j_2,j_3})^2 \\ +(\beta_{j_1,j_2,j_3+1} - \beta_{j_1,j_2,j_3})^2 \end{array}},$$

in which $p_1 = 4$, $p_2 = 3$ and $p_3 = 2$. In total, there are as many groups as 3-dimensional voxels, there are then $G = p_1 p_2 p_3 = 24$ groups.

We thus construct the A matrix to reflect this penalty. The first group is

$$A_1 = \begin{bmatrix} \mathbf{-1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{-1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{-1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Thus, for matrix A_g (in group g) we have a -1 in the gth column in all dimensions, a 1 in the $(g+1)$th column for the 1st dimension, a 1 in the $(p_1 + g)$th column for the 2nd dimension

and a 1 in the $(p_1 \cdot p_2 + g)$th column for the 3rd dimension. Note that when these indices fall outside of the A matrix (i.e., the indices are greater than p_1, p_2, or p_3, respectively) then the whole row (but not the group!) must be set to zero (or handled in some other way not specified here).

We have

$$\partial TV(\beta) = A^\top \begin{bmatrix} \partial\|A_1\beta\|_2 \\ \vdots \\ \partial\|A_{24}\beta\|_2 \end{bmatrix} \tag{26}$$

and use Equation 6 to obtain

$$\partial\|A_g\beta\|_2 \ni \begin{cases} \frac{A_g\beta}{\|A_g\beta\|_2} & \text{if } \|A_g\beta\|_2 > 0, \\ \frac{\alpha u}{\|u\|_2},\ \alpha \sim \mathcal{U}(0,1), u \sim \mathcal{U}(-1,1)^3 & \text{if } \|A_g\beta\|_2 = 0. \end{cases} \tag{27}$$

We note that A is very sparse, which greatly helps to speed up and save memory in the implementation.

We now illustrate how to use **pylearn-simulate** to generate the A matrix.

```
>>> import simulate
```

The size of the underlying $2 \times 3 \times 4$ image is then defined as

```
>>> shape = (2, 3, 4)
```

Defining the shape allows then to generate the A matrix:

```
>>> A = simulate.functions.TotalVariation.A_from_shape(shape)
>>> print A
>>> print A[0].toarray()[0,:], "\n", \
...   A[1].toarray()[0,:], "\n", \
...   A[2].toarray()[0,:]
```

The output of the above code is:

```
[<24x24 sparse matrix of type '<type 'numpy.float64'>'
    with 36 stored elements in Compressed Sparse Row format>,
 <24x24 sparse matrix of type '<type 'numpy.float64'>'
    with 32 stored elements in Compressed Sparse Row format>,
 <24x24 sparse matrix of type '<type 'numpy.float64'>'
    with 24 stored elements in Compressed Sparse Row format>]

[-1.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
[-1.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
[-1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

The function `simulate.functions.TotalVariation.A_from_shape` thus returns a 3-vector with a sparse matrix in each element. This is for computational reasons (to avoid looping over p groups). The first element of `A` contains the first dimension, the second element the

second dimension and the third element the third dimension. We note that the first group (as seen in the second print) is precisely as given above.

B.2. Overlapping group LASSO

Gradient of non-smooth overlapping group LASSO

The group LASSO penalty is defined as

$$\mathrm{GL}(\beta) = \sum_{g=1}^{G} \eta_g \|\beta_g\|_2, \tag{28}$$

where η_g is a group weight that accounts for varying group sizes, β_g is a sub-vector of β. Note that any two sub-vectors β_a and β_b are allowed to overlap, i.e., it is possible that a variable β_j is present in both subvectors β_a and β_b.

We will define matrices A_g, one for each group, such that

$$\mathrm{GL}(\beta) = \sum_{g=1}^{G} \eta_g \|\beta_g\|_2 = \sum_{g=1}^{G} \|A_g \beta\|_2. \tag{29}$$

We will illustrate the construction of A_g by an example. We are working with 6 variables, i.e., $\beta \in \mathbb{R}^6$. We define two groups, such that variables 1, 3, 4 and 6 belong to group 1 and variables 2, 4 and 6 belong to group 2. We define A_1 as

$$A_1 = \eta_1 \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

and A_2 as

$$A_2 = \eta_2 \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The matrices A_g are thus $|g| \times p$, where $|g|$ is the number of variables in group g. Each row of A_g selects one variable to the group; if variable β_j belongs to group g, then A_g has a row with all zeros except for the element in the jth position which is η_g. Thus, A_g selects the variables that belong to group g and thus $\eta_g \beta_g = A_g \beta$.

We again use Theorem 4.3 and obtain

$$\partial GL(\beta) = A^\top \begin{bmatrix} \partial \|A_1 \beta\|_2 \\ \vdots \\ \partial \|A_G \beta\|_2 \end{bmatrix}.$$

We now illustrate how to use **pylearn-simulate** to generate the A matrix for overlapping group LASSO.

```
>>> import simulate
```

First, we define the number of variables and the groups:

```
>>> p = 6
>>> groups = [[0, 2, 3, 5], [1, 3, 5]]
```

Then, the linear operator (the matrix A) for overlapping group LASSO is generated:

```
>>> A = simulate.functions.SmoothedGroupLasso.A_from_groups(p, groups,
...   weights = [3.14159, 2.71828])
>>> print A
>>> print A[0].toarray()
>>> print A[1].toarray()
```

The output of the above code is:

```
[<4x6 sparse matrix of type '<type 'numpy.float64'>'
    with 4 stored elements in Compressed Sparse Row format>,
 <3x6 sparse matrix of type '<type 'numpy.float64'>'
    with 3 stored elements in Compressed Sparse Row format>]

[[ 3.14159  0.       0.       0.       0.       0.     ]
 [ 0.       0.       3.14159  0.       0.       0.     ]
 [ 0.       0.       0.       3.14159  0.       0.     ]
 [ 0.       0.       0.       0.       0.       3.14159]]

[[ 0.       2.71828  0.       0.       0.       0.     ]
 [ 0.       0.       0.       2.71828  0.       0.     ]
 [ 0.       0.       0.       0.       0.       2.71828]]
```

The function `simulate.functions.SmoothedGroupLasso.A_from_groups` thus returns a vector with two sparse matrices. The elements of this vector are the A matrices for each group. We note that the printed A matrices correspond to those in the example above. Note also that the variable indices are zero-based.

Gradient of smoothed group LASSO

The gradient of Nesterov-smoothed complex penalties is given in Equation 7. Thus, in order to compute the gradient we need the linear operator A and the dual variable α^*. The A matrix is described above; the computation of the dual variable for group LASSO is given by Chen and Liu (2011), but without derivation. We therefore derive the solution here.

We compute the parts of the dual variable that corresponds to each group, i.e., α_g^*. We

express α_g^* as a projection of the vector $\frac{1}{\mu}A_g\beta$ onto the compact space K_g, i.e.,

$$\begin{aligned}
\alpha_g^* &= \arg\max_{\alpha \in K_g} \left\{ \langle \alpha_g, A_g\beta \rangle - \frac{\mu}{2}\|\alpha_g\|_2^2 \right\} \\
&= \arg\min_{\alpha \in K_g} \left\{ \|\alpha_g\|_2^2 - \frac{2}{\mu}\langle \alpha_g, A_g\beta \rangle \right\} \\
&= \arg\min_{\alpha \in K_g} \left\{ \left\| \alpha_g - \frac{A_g\beta}{\mu} \right\|_2^2 - \left\| \frac{A_g\beta}{\mu} \right\|_2^2 \right\} \\
&= \arg\min_{\alpha \in K_g} \left\{ \left\| \alpha_g - \frac{A_g\beta}{\mu} \right\|_2^2 \right\} \\
&= \mathrm{proj}_{K_g}\left(\frac{A_g\beta}{\mu} \right),
\end{aligned} \tag{30}$$

where K_g is the unit ℓ_2 ball. Thus, the projection operator is

$$\mathrm{proj}_{K_g}(x) = \mathrm{proj}_{\ell_2}(x) = \begin{cases} \frac{x}{\|x\|_2}, & \text{if } \|x\|_2 > 1, \\ x, & \text{otherwise.} \end{cases} \tag{31}$$

The dual variable, α^*, is then the concatenation of the sub-vectors of all groups.

Affiliation:

Tommy Löfstedt
Brainomics Team
Neurospin
CEA Saclay
91190 Gif-sur-Yvette, France
E-mail: lofstedt.tommy@gmail.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
October 2018, Volume 87, Issue 3 *Submitted:* 2014-12-28
doi:10.18637/jss.v087.i03 *Accepted:* 2016-09-07