Fused Lasso in Two Dimensions and Image Denoising

STAT 771 - Final Project

Sean Kent

Introduction

It would be hard to overstate the popularity of the lasso variation on the classical linear regression, where a penalty term is added to the minimization problem that limits the number and magnitude of coefficients. This model works well empirically in the $p \gg n$ problem, where the number of model parameters (covariates) is much larger than the sample data. For a regression $y_i = \sum_{j=1}^{p} x_{ij}\beta_j, i = 1, \ldots, n$, the lasso solution can be expressed as $\hat{\beta}$ that is the minimum of $f(\beta)$ in the following Lagrange form:

$$f(\beta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda_1 \sum_{j=1}^{p} |\beta_j|,$$

where λ_1 is a hyperparameter that determines how much the penalty term dominates. In 2005, The so-called fused lasso was proposed (Tibshirani et al. 2005), with the following objective function:

$$f(\beta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda_1 \sum_{j=1}^{p} |\beta_j| + \lambda_2 \sum_{j=2}^{p} |\beta_j - \beta_{j-1}|$$

The fused lasso model considers the additional penalty term with hyperparameter λ_2, which acts to restrict the difference in coefficients β_j from coefficients near to it in the model. Because this penalty is a convex contraint, there is still a unique minimum $\hat{\beta}$. In an ordinary regression this term would be meaningless–there is no natural order to the coefficients. However, this model has been proposed in cases where 'nearby' β terms correspond to a natural ordering.

This paper will have two related goals. First, to describe in some detail the fused lasso, some common generalizations, and what algorithms are currently used to fit the model. After adequate coverage of the fused lasso, this paper will focus on one application described in (Friedman et al. 2007): image denoising. In particular, we will compare the performance of the fused lasso in image denoising to that of block matching and 3D filtering (BM3D), a popular algorithm that is quite powerful at image denoising in practice.

Fused Lasso

While the additional penalty term in the objective function of the fused lasso has some uses, the restriction to one dimension makes applications tough and somewhat rare. This yields a few natural extentions, where the sum in the penalty term on the difference of parameters β_j is taken not simply over the ordering of the model covariates, but rather over a graph (where the sum is over edges that connect a vertex, and each vertex has an associated β_j), or over a multi-dimensional grid (where the sum of over adjacent points in the grid, and each point has an associated β_j) (Xin et al. 2016). Through these extensions, the fused lasso has been used to diagnose Alzheimer's (Xin et al. 2016), detect hot spots in comparative genomic hybridization (Tibshirani and Wang 2008), and even smoothing images (Friedman et al. 2007). Their general forms are given below

- Fused Lasso over a Graph G with edges E:

$$f(\beta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda_1 \sum_{j=1}^{p} |\beta_j| + \lambda_2 \sum_{(i,j)\in E} |\beta_i - \beta_j|$$

- Fused Lasso over a grid G of pixels $p = (i, j)$, $i = 1, \ldots, n_1$, $j = 1, \ldots, n_2$:

$$f(\beta) = \frac{1}{2}\sum_{i=1}^{n_1}\sum_{j=1}^{n_2}(y_{i,j} - \beta_{i,j})^2 + \lambda_1 \sum_{i=1}^{n_1}\sum_{j=1}^{n_2}|\beta_{i,j}| + \lambda_2 \left[\sum_{i=2}^{n_1}\sum_{j=1}^{n_2}|\beta_{i,j} - \beta_{i-1,j}| + \sum_{i=1}^{n_1}\sum_{j=2}^{n_2}|\beta_{i,j} - \beta_{i,j-1}|\right]$$

All of the objective functions given by generalizations of the fused lasso are sums of convex functions (either parabolas or absolute values), and so they are convex. It is trivial to show they admit a unique minimum. However, as in the case of the lasso, the constraints do not admit nice derivatives, and so there is no formula to find this minimum. Various algorithms must be used in order to quickly achieve this solution in a variety of contexts. For the fused lasso and its generalizations, Xin et al. (2016) give a summary of existing algorithms. Two of primary interest are summarized below:

- The original fused lasso paper proposed an algorithm based on Stanford's SQOPT which solves large scale linear and quadratic programming problems with constraints, although it doesn't scale well to large dimensions (Tibshirani et al. 2005). In image smoothing, the dimensions can quickly get large and overwhelm this algorithm.
- Another algorithm based on an extension of path-wise coordinate descent was proposed in the special case when the design matrix is an identity matrix, which is quite fast, even when the number of dimensions is large (Friedman et al. 2007). This algorithm has been adapted for image smoothing and showed good speed results and convergence.

Coordinate Descent Algorithm

The path-wise coordinate descent algorithm for the fused lasso problem developed by Friedman et al. (2007) works as follows:

Given a grid G of pixels $p = (i, j)$, $i = 1, \ldots, n_1$, $j = 1, \ldots, n_2$, we seek to minimize the objective function $f(\beta)$ where $y_{ij} = y_p$ is the data value at point p. The penalty term with λ_2 can be thought of as a double sum, first taking the sum over all points, and at each point, adding up $|\beta_p - \beta_{p'}|$ for all points p' that are touching p.

Following the notation in Friedman et al. (2007), if we partition the grid into K groups $\{G_k\}$, and constrain the estimates in each group to one unique estimate, i.e. $\forall p \in G_k, \beta_p = \gamma_k$, then the solution to the objective function $f(\beta)$ is the solution to the following objective function (in γ_k):

$$h(\gamma) = \frac{1}{2}\sum_{k=1}^{K} N_k(\bar{y}_k - \gamma_k)^2 + \lambda_1 \sum_{k=1}^{K} N_k|\gamma_k| + \frac{\lambda_2}{2}\sum_{D(k,k')=1} w_{kk'}|\gamma_k - \gamma_{k'}|$$

where N_k is the number of pixels in G_k, $\bar{y}_k$ is the mean pixel value for those in G_k, $D(k, k') = 1$ represents the set of k and k' that are touching in the image, and $w_{kk'}$ is the number of adjacent pixels between groups G_k and $G_{k'}$. To minimize $h(\gamma)$,

Initialize

- Fix λ_1 as given
- Set $\lambda_2 = 0$
- Start the set of groups G_k as individual pixels
- Get initial solution by soft-thresholding $\gamma_k = S(\bar{y}_k, \lambda_1) = sign(\bar{y}_k)(|\bar{y}_k| - \lambda_1)_+$.

Iterate

Increment λ_2 by some small $\delta > 0$

For $k = 1, 2, \ldots, K$,

(a) *Descent*: Minimize $h(\gamma)$ in γ_k, holding all other $\gamma_{k'}$ constant. Details on this step are given below.

(b) *Fusion*: If the solution in (a) does not change the value of γ_k, consider a possible fusion of the group G_k with one other group $G_{k'}$ touching G_k. Then minimize $h(\gamma)$ in $\gamma_m = \gamma_k = \gamma_{k'}$, with the new group $G_m = G_k \cup G_{k'}$, holding all other values constant. If a fusion successfully improves the objective function, we accept the new grouping and estimate. Otherwise, continue to try fusion with all other groups $G_{k'}$ touching G_k. If none are successful, γ_k remains the same.

Increment k by 1.

Repeat (a) and (b) (called a cycle) until a cycle fails to change any γ_k (up to some small tolerance). This estimate is the solution for the current value of λ_2.

(c) *Smoothing*: Increment λ_2 by some small $\delta > 0$, and repeat sets of cycles until the maximum desired value of lambda is achieved.

Descent step in detail: Holding all but one parameter values γ_k constant, minimization of $h(\gamma)$ is somewhat straightforward. Friedman et al. (2007) shows that the derivative of $h(\gamma)$ is piecewise linear, and has breaks at $\{\gamma_{k'}\}$ where k' indexes a set touching G_k. Hence, we can check for a zero crossing at points just above and just below $\{\gamma_{k'}\}$. If zero is crossed, meaning we have a minimum of the $h(\gamma)$, linearity makes finding that minimum trivial. Set the new γ_k to be the point minimizing $h(\gamma)$. Otherwise, if zero is never crossed, we check the objective function locally, and set γ_k as the minimum of $h(\gamma)$ over the set of discontinuities.

For sufficiently small delta, this algorithm usually converges to the unique minimum of $f(\beta)$. Friedman et al. (2007) prove its convergence for the one dimensional case, but only observe that it works well in practice for the two dimensional case.

Image Smoothing

Image smoothing is a notoriously complex task, with a wide variety of approaches and algorithms to attempt to solve it. These algorithms range from very basic to extremely complicated. On the basic side, a smoothed pixel may simply be an average or weighted average of the pixels surrounding it; the complexity of other algorithms is nearly unbounded. On the surface, fused lasso seems like a natural application to image smoothing, where the (i, j)th pixel value corresponds to a parameter β_{ij}, and we seek to smooth this potentially noisy pixel by estimating a $\hat{\beta}_{ij}$ that is often similar to the surrounding pixels in the non-noisy image. At the end of the day, we are minimizing a sum squared error criteria, with a penalty on how different nearby pixels *should* be, which we hope will reduce any inherent noise this image might have.

In the computer science realm, the block matching and 3D filtering (BM3D) algorithm is one of the top image denoising algorithms, which shows great results in practice (Dabov et al. 2007). While the details of this algorithm are not the focus of this paper, the main ideas of it are quite intuitive. Figure 1 (taken from Dabov et al. (2007)) outlines the main steps of the algorithm. In both Step 1 and Step 2, the first substep is the grouping by block matching. Instead of simply looking at the pixels neighboring a point of interest (often called a window around the point p), this substep scans the image for other windows (perhaps not centered on neighboring points) that are similar (using a distance metric based on the ℓ^2 norm) and then combines the information from all of these windows, rather than just the one centering our point. In this way, the algorithm is able to reduce a lot of the noise from the data beforming any smoothing. The second key element of this algorithm comes from the collaborative Weiner filtering which minimizes the mean square error between the estimated random process and the desired process (our image). Given that both BM3D and fused lasso follow some sort of minimization of MSE, the question must be asked: which performs better? If fused lasso can stack up to the BM3D algorithm–both in terms of accuracy to the orignal image and speed–it would gain a lot of recognition as a top image smoothing algorithm.

In order to make a comparison between the fused lasso and the BM3D algorithm, we will need a data set by which to evaluate both algorithms on. Many papers in image denoising use the University of Southern California's Signal and Image Processing Institute's (SIPI) picture data set, and a portion of this is what the original paper on BM3D was evaluated on. As a test set, this paper will use four 256 by 256 images that

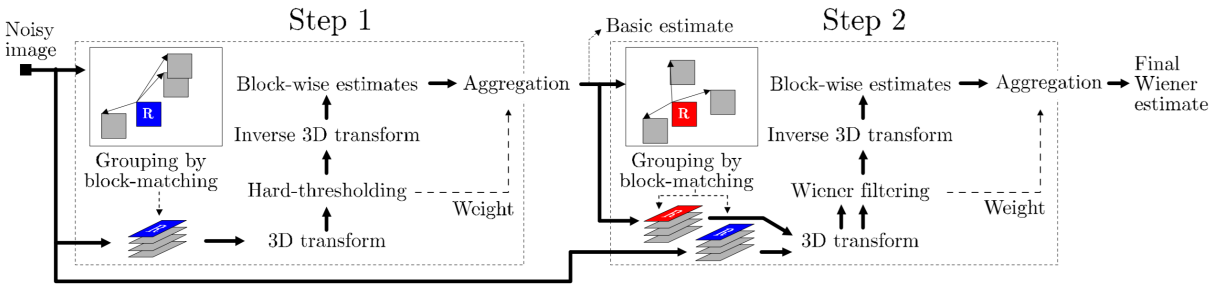

Figure 1: A summary of the BM3D algorithm's main steps

make up a subset from Dabov et al. (2007). Friedman et al. (2007) use an image of Sir Ronald Fisher to demonstrate the effectiveness of fused lasso on the task of image smoothing, but unfortunately that image was not readily available for comparison.

Evaluation and Results

A typical metric for evaluating the quality of an image denoising is the Peak Signal-to-Noise Ratio (PSNR). This metric compares the true image y to the estimated image $\hat{y}$, by the following formula:

$$PSNR\left(\hat{y}\right) = 10\,\log_{10}\left(\frac{255^2}{|P|^{-1}\sum_{(i,j)}\left(y_{ij}-\hat{y}_{ij}\right)^2}\right),$$

where y_{ij} is the pixel for image y at the (i,j)th pixel position and $|P|$ is the number of pixels, and the image follows a greyscale with a maximum pixel value of 255 (following the standard RGB format). Observe that PSNR is essentially an inverted metric of Mean Square Error (MSE), that is scaled. If an estimated image is close to the true non-noisy image, then the PSNR metric will be large, indicating the estimate captures much of the signal in the true image.

Code was written in R to implement the fused lasso algorithm in the case of a 2d grid of points; it is provided in the appendix. While this code works reasonbly fast on small images, it is somewhat slow compared to the R implementation in the `flsa` package. To demonstrate the effectiveness of the hand-written code, a small toy example is provided in the following text. The `flsa` package will be used in the final comparison of fused lasso to BM3D.

Toy Example

While pictures tend to be quite complex, it serves to see whether fused lasso can smooth an extremely simple image, that of a 16 by 16 pixel plus sign.

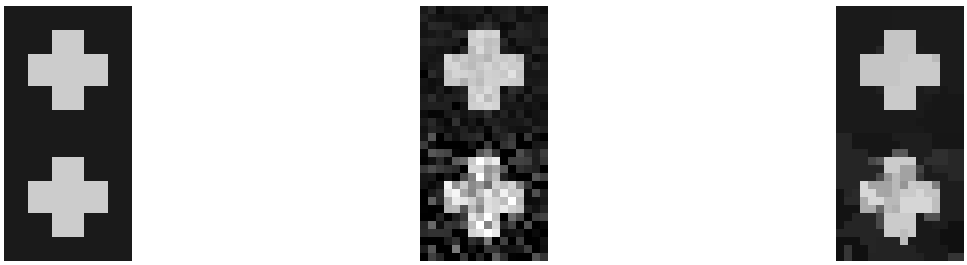

Figure 2: [Right to Left] Original cross image, noisy image, fused lasso estimate.
[Top to Bottom] Gaussian noise with standard deviation 10 and 40

Upon visual inspection of Figure 2, the fused lasso is performing quite well in smoothing the noisy cross image to restore it to the original. In the case when only Gaussian noise with standard devation 10 (on an image scale of 0 to 255), the image is almost completely restored. A comparison of PSNR shows that the fused lasso brings the standard deviation 10 image from 28.05 to 37.53. For the standard deviation 40 image, the PSNR goes from 17.41 to 21.46

Test Set of Images

In our test set, we consider the four 256 by 256 pixel greyscale images called "Cameraman256", "house", "montage", and "peppers256". To each image, Gaussian noise of standard deviation 20, 40, and 100 (relative to an image with values in 0 to 255) is added, and both BM3D and fused lasso are applied to the image. BM3D has implementation in MatLab, which is availible at http://www.cs.tut.fi/~foi/GCF-BM3D/, a site by the authors of the original Bm3D paper (Dabov et al. 2007). For each estimated image, we compute the PSNR. It is also important to evaluate these images visually, and so Figure 3 shows a comparision of fused lasso and BM3D for the house image at various noise levels.

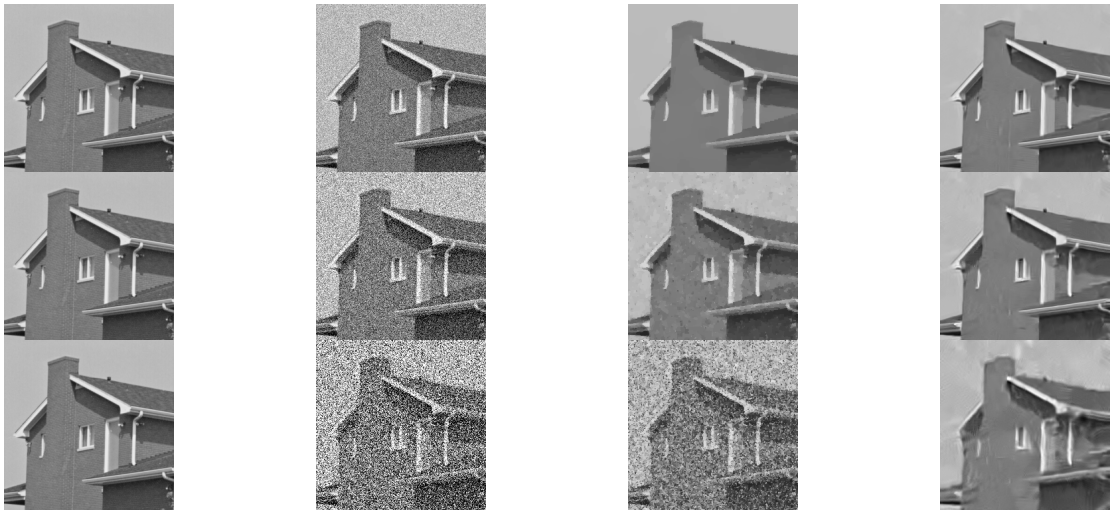

Figure 3: [Right to Left] Original cross image, noisy image, fused lasso estimate, BM3d estimate.
[Top to Bottom] Gaussian noise with standard deviation 20, 40, and 100

From Figure 3 we can see that the BM3D algorithm is far superior based on the visualization test, although the fused lasso is quite impressive in smoothing the image, for reasonable levels of noise (especially at SD of 40). The image with SD 100 noise is almost unreadable, but after smoothing at least the fused lasso can highlight some of the key features of the house. One other observation of note is that the fused lasso can go too far in the smoothing, erasing some of the key features of the chimney. Since fused lasso requires tuning of hyperparameters, it is possible that a better choice would lead to better results, but each image was tested for a number of λ_2, with $\lambda_1 = 0$, and the optimal image was chosen as that which maximizes PSNR (note this does 'peek' at the true image, but the goal was to give fused lasso the best chance at outperforming BM3d; optimal hyperparameters are usually chosen via cross validation).

When comparing all four test images on the PSNR metric, Table 1 again shows the BM3d algorithm to be far superior to fused lasso. Especially in the case of large noise, the fused lasso is simply not smoothing the image towards the true image as much as BM3D. Again, for smaller noise levels, the fused lasso is able to capture a decent amount of the signal from the image, but this is not the case for larger noise levels.

Conclusion

The fused lasso and its generalizations have been used to some degree in applications (Xin et al. 2016, Tibshirani and Wang (2008), fhht07). In the context of image denoising, this paper shows that while the fused

Table 1: PSNR of Fused Lasso and BM3D

Image [SD of Noise]	Noisy Image	Fused Lasso	BM3D
Cameraman			
20	22.44	26.45	30.48
40	16.64	24.92	27.17
100	10.29	16.57	23.07
House			
20	22.13	30.11	33.76
40	16.33	27.96	30.64
100	10.17	17.23	25.87
Montage			
20	22.56	29.04	33.59
40	17.01	26.32	29.52
100	10.52	16.58	23.88
Peppers			
20	22.21	27.69	31.28
40	16.43	25.82	27.70
100	10.21	16.88	23.39

lasso can perform image smoothing for small levels of additional noise, it cannot compete with state-of-the-art algorithmss like BM3D. Given that both algorithms run in similar time, it stands to reason that BM3D would be preferred in most circumstances (unless the user exclusively wanted to use R, for which there is no current BM3D implementation). It seems that the block matching–which gathers information from more than just nearby pixels–is increasing the information on which to smooth the image, which is resulting in better performance. The coordinate descent algorithm allows for reasonable computation time, even with large images (256 by 256 pixels), but may be more appropriate for models in which there is more than one data point for each parameter (as is usually the case for regression problems). In image smoothing, we only have one point and it's neighbors to gather information from for each model parameter. Future work may consider a similar block-matching structure in the fused lasso, specifically for the problem of image smoothing.

Appendix: R Code

```
###-------------------------------------------------------###
###---- Setup                                             ----
###-------------------------------------------------------###
{
library(tidyverse)
library(png)
par(pty="s")
}
###-------------------------------------------------------###
###---- Images for testing                                ----
###-------------------------------------------------------###
{
image_noise <- function(image, sd, seed=8) {
  ## inputs:  image, a matrix with entries in [0.1] representing a greyscale image
  ##          sd, a standard deviation which determines the noise added.
  ##            Scale can be given based a grey scale of 255
  ## output: image + gaussian noise
```

```
  ## adjust sd to appropriate scale
  if ( sd > 1 ) {
    sd <- sd/255
  }

  set.seed(seed)
  noisy <- image + sd * matrix( rnorm(prod(dim(image))), nrow = dim(image)[1])

  ## fix values between 0 and 1
  noisy[noisy < 0] <- 0
  noisy[noisy > 1] <- 1
  return( noisy )
}

## small (16x16) toy example
cross <- matrix(0.1, nrow=16, ncol=16)
cross[7:10,4:13] <- 0.8
cross[4:13,7:10] <- 0.8
cross_noi_s10 <- image_noise(cross, 10)
cross_noi_s40 <- image_noise(cross, 40)

## read in images from the BM3D paper (standard in image smoothing)
image_names <- c("Cameraman256", "house", "montage", "peppers256")

all_true_images <- lapply(image_names, FUN = function(x) {
  image <- readPNG( paste0("./BM3D_images/",x,".PNG") )
})

all_noisy_images <- lapply(image_names, FUN = function(x) {
  image <- readPNG( paste0("./BM3D_images/",x,".PNG") )
  s20 <- image_noise(image, 20)
  s40 <- image_noise(image, 40)
  s100 <- image_noise(image, 100)
  return(list(s20=s20, s40=s40, s100=s100))
})

names(all_noisy_images) <- names(all_true_images) <- image_names
}
###-------------------------------------------------------###
###---- Functions                                        ----
###-------------------------------------------------------###
{
soft_threshold <- function(image, penalty) {
  sign(image) * max( abs(image)-penalty, 0 )
}

find_d1_groups <- function(groups_, group_one_) {
  ## returns groups that are touching the selected group

  nrow <- dim(groups_)[1]
  ncol <- dim(groups_)[2]
  ind <- which(groups_ == group_one_)

  d1_groups_ind <- c(ind[row(groups_)[ind] != nrow] + 1, # index below all not in last row
                     ind[row(groups_)[ind] != 1]    - 1, # above all not in first row
                     ind[col(groups_)[ind] != ncol] + nrow, # not right of last col
                     ind[col(groups_)[ind] != 1]    - nrow) %>% # not left of 1st col
    setdiff(ind) %>%
```

```
    unique()

  out <- groups_[d1_groups_ind]

  return(out)
}

fuse_groups <- function(groups_, group_one_, group_two_) {
  groups_[groups_==group_two_] <- group_one_
  return(groups_)
}

w <- function(groups_, group_one_, group_two_) {
  ind_one <- which(groups_==group_one_)
  ind_two <- which(groups_==group_two_)
  nrow <- dim(groups_)[1]

  ind_d1_to_one <- c(ind_two+1, ind_two-1, ind_two+nrow, ind_two-nrow)
  return(sum(ind_one %in% ind_d1_to_one))
}

objective_function <- function(image_, estimate_, groups_, lam1_, lam2_) {
  mse <- 1/2  * sum( (image_ - estimate_)^2 )
  l1 <- lam1_ * sum( abs(estimate_) )

  fused <- lam2_/2 * sum( sapply( unique(as.vector(groups_)), FUN = function(k) {
    ## first, sum over each group k
    d1_groups <- groups_ %>% find_d1_groups(k)
    gamma_k <- estimate_[which(groups_==k)][1]

    out_k <- sum( sapply(d1_groups, FUN = function(k_prime) {
      ## second, sum over k_prime that are distance 1 from k
      gamma_k_prime <- estimate_[which(groups_==k_prime)][1]

      out_k_prime <- w(groups_, k, k_prime) * abs( gamma_k - gamma_k_prime )
      return(out_k_prime)
    }))
    return(out_k)
  }))

  out <- mse + l1 + fused
  return(out)
}

local_objective_function <- function(k, image_, estimate_, groups_, lam1_, lam2_) {
  mse <- 1/2  * sum( (image_[groups_==k] - estimate_[groups_==k])^2 )
  l1 <- lam1_ * sum( abs(estimate_[groups_==k]) )

  d1_groups <- groups_ %>% find_d1_groups(k)
  gamma_k <- estimate_[which(groups_==k)][1]

  fused <- lam2_/2 * sum( sapply(d1_groups, FUN = function(k_prime) {
    ## second, sum over k_prime that are distance 1 from k
    gamma_k_prime <- estimate_[which(groups_==k_prime)][1]

    out_k_prime <- w(groups_, k, k_prime) * abs( gamma_k - gamma_k_prime )
    return(out_k_prime)
  }))
```

```
  out <- mse + l1 + fused
  return(out)
}

gradient_objective_function_k <- function(k_, gamma_k_=NULL, image_, estimate_, groups_,
                                          lam1_, lam2_) {
  ## k_ is group of interest, gamma_k_ can be set as any estimate
  ## image_: original image; estimate_: current estimate (gives gamma_k's)
  if(is.null(gamma_k_)) {
    gamma_k_ <- estimate_[groups_==k_]
  }

  mse <- -sum( (image_[groups_==k_] - gamma_k_) ) # \bar{y_k} - gamma_k
  l1 <- lam1_ * sum( sign(gamma_k_) )

  d1_groups <- groups_ %>% find_d1_groups(k_)

  fused <-  lam2_/2 * sum( sapply(d1_groups, FUN = function(k_prime) {
    ## sum over k_prime that are distance 1 from k
    gamma_k_prime <- estimate_[which(groups_==k_prime)][1]

    out_k_prime <- w(groups_, k_, k_prime) * sign( gamma_k_ - gamma_k_prime )
    return(out_k_prime)
  }))


  out <- mse + l1 + fused
  return(out)
}

find_gamma_k <- function(k_, image_, estimate_, groups_, lam1_, lam2_) {
  ## minimize objective function over gamma_k, holding all other gamma_k_prime as fixed
  ## This utilizes the fact that the gradient of the objective function is
  ##  piecewise linear with breaks at 0 and gamma estimates of distance 1 neighbors.

  d1_groups_ <- groups_ %>% find_d1_groups(k_) %>% unique()
  gamma_k_prime_ <- unique(estimate_[groups_ %in% d1_groups_])
  eps <- 1e-4
  breaks_ <- sort( c( gamma_k_prime_-eps, gamma_k_prime_+eps, 0) )

  gradient_at_breaks_ <- sapply(breaks_, FUN = gradient_objective_function_k,
                                k_=k_, image_=image_, estimate_=estimate_,
                                groups_=groups_, lam1_=lam1_, lam2_=lam2_)

  ## calculate min of objective function
  if(sum(unique(sign(gradient_at_breaks_))) == 0) { # gradient crosses zero

    ind <- max( which(gradient_at_breaks_ < 0) )  # index before gradient crosses zero

    # x = - y_1 * (x_2 - x_1) / (y_2 - y_1) + x_1 gives x at which line hits 0
    find_zero <- function(x,y) {
      slope <- (y[2] - y[1]) / (x[2] - x[1])
      zero <- - y[1] / slope + x[1]
      return(zero)
    }

    min_ <- find_zero(breaks_[ind:(ind+1)], gradient_at_breaks_[ind:(ind+1)])
```

```
  } else { # gradient doesn't cross zero

    ## Make more efficient by taking advantage of (39)!!!!!!
    estimate_w_k_prime_ <- lapply(d1_groups_, FUN = function(k_prime_) {
      estimate_[groups_==k_prime_][1] * (groups_ == k_) + estimate_ * (groups_ != k_)
    })
    estimate_w_0 <- 0 * (groups_ == k_) + estimate_ * (groups_ != k_)

    ## take gamma_k_prime with smallest objective function
    obj_at_breaks_ <- sapply(estimate_w_k_prime_, FUN = local_objective_function,
                             k=k_, image_=image_, groups_=groups_,
                             lam1_=lam1_, lam2_=lam2_)
    obj_at_zero <- local_objective_function(k_, image_, estimate_w_0, groups_,
                                            lam1_, lam2_)

    if ( min(obj_at_breaks_) < obj_at_zero ) {
      k_min <- d1_groups_[obj_at_breaks_ == min(obj_at_breaks_)]
      min_ <- estimate_[groups_==k_min][1]
    } else {
      min_ <- 0
    }
  }
  return(min_)
}

diff_in_objective_function <- function(gamma_m_, k_, k_prime_, image_, estimate_,
                                       groups_, lam1_, lam2_) {
  gamma_k_ <- estimate_[groups_==k_][1]
  gamma_k_prime_ <- estimate_[groups_==k_prime_][1]
  n_k_ <- sum(groups_==k_)
  n_k_prime_ <- sum(groups_==k_prime_)
  y_k_ <- mean(image_[groups_==k_])
  y_k_prime_ <- mean(image_[groups_==k_prime_])
  ##w_kl_ <- w(groups_, k_, )

  n_m_ <- n_k_ + n_k_prime_
  y_m_ <- (n_k_*y_k_ + n_k_prime_*y_k_prime_) / n_m_

  mse_diff <- (-n_k_*(y_k_ - gamma_k_)^2 +
                 -n_k_prime_*(y_k_prime_ - gamma_k_prime_)^2 +
                 n_m_*(y_m_ - gamma_m_)^2 )

  l1_diff <- lam1_ * (-n_k_*abs(gamma_k_) +
                        -n_k_prime_*abs(gamma_k_prime_) +
                        n_m_*abs(gamma_m_) )

  fused_k <- function(l,k,gamma_k,groups) {
    ## for given k, and l with d(k,l)=1, finds fused penalty component
    gamma_l <- estimate_[which(groups==l)][1]

    out_k_prime <- w(groups, k, l) * abs( gamma_k - gamma_l )
    return(out_k_prime)
  }

  d1_groups_k <- find_d1_groups(groups_, k)
  d1_groups_k_prime <- find_d1_groups(groups_, k_prime_)
  d1_groups_m <- union(d1_groups_k, d1_groups_k_prime) %>% setdiff( c(k_,k_prime_) )
  groups_fused_ <- fuse_groups(groups_, k_, k_prime_)
```

```
  fused_diff <- lam2_/2 * ( -sum( sapply(d1_groups_k, FUN = fused_k,
                                         k=k_, gamma_k=gamma_k_, groups=groups_) ) +
                              -sum( sapply(d1_groups_k_prime, FUN = fused_k,
                                           k=k_prime_, gamma_k=gamma_k_prime_,groups=groups_) ) +
                              sum( sapply(d1_groups_m, FUN = fused_k,
                                          k=k_, gamma_k=gamma_k_, groups=groups_fused_) ) )

  return(mse_diff + l1_diff + fused_diff)
}
}
###-------------------------------------------------------###
###---- Main Algorithm                                  ----
###-------------------------------------------------------###

fused_lasso <- function(image, lam1, lam2=0, lam2_max, delta, tol=5e-4, verbose=FALSE) {

  ## start groups as all separate
  groups <- matrix(1:prod(dim(image)), nrow=dim(image)[1])

  estimate_list <- list()
  i <- 1

  while (lam2 < lam2_max) {
    if (lam2 == 0) {
      ## set initial estimate to soft-thresholded value of image
      estimate <- soft_threshold(image, lam1)
      estimate_list[[i]] <- estimate # record the estimate
      i <- i + 1
      lam2 <- lam2 + delta
    }

    if(verbose) { cat("\n Lambda2:",lam2,"\n") }
    gamma_change <- TRUE
    while (gamma_change) { # keep track of whether we change any gamma

      gamma_k_change <- logical(length(unique(as.vector(groups))))

      for (k in unique(as.vector(groups))) {
        gamma_k_change[k] <- FALSE # start with no change

        if ( !(k %in% as.vector(groups)) ) next # needed if we fuse a group

        ################.
        ## Descent
        gamma_k_new <- find_gamma_k(k_ = k, image, estimate, groups, lam1, lam2)
        gamma_k <- estimate[groups==k][1]

        if ( abs(gamma_k_new - gamma_k) > tol ) {
          estimate[groups == k] <- gamma_k_new
          gamma_k_change[k] <- TRUE

        } else { # gamma_k_new == gamma_k
          ################.
          ## Fusion
          d1_groups <- find_d1_groups(groups, k)
          #obj <- objective_function(image, estimate, groups, lam1, lam2)

          for (k_prime in d1_groups) {
```

```
            groups_tmp <- fuse_groups(groups, k, k_prime) # provisionally fuse k and k'
            gamma_m <- find_gamma_k(k_ = k, image, estimate, groups_tmp, lam1, lam2)
            if (is.na(gamma_m)) {
              break
            }

            estimate_tmp <- estimate
            estimate_tmp[groups %in% c(k,k_prime)] <- gamma_m

            diff <- local_objective_function(k, image, estimate_tmp, groups_tmp, lam1, lam2) -
              local_objective_function(k, image, estimate, groups, lam1, lam2) -
              local_objective_function(k_prime, image, estimate, groups, lam1, lam2)
            improved_criterion <- diff < 0

            if (improved_criterion) {
              groups <- groups_tmp
              estimate[groups_tmp %in% c(k,k_prime)] <- gamma_m
              gamma_k_change[k] <- TRUE
              break # ends for k_prime
            } # else keep gamma_k the same

          } # end for k_prime

        } # end else


      } # end for k


      gamma_change <- sum(gamma_k_change, na.rm = TRUE) > 0 # Any gamma_k change?

      ## Display prevalent metrics
      if(verbose) {
        if (prod(dim(image)) < 300) {
          obj <- objective_function(image, estimate, groups, lam1, lam2)
        }
        n_groups <- length(unique(as.vector(groups)))
        cat("  Cycle completed.")
        if (prod(dim(image)) < 300) { cat(" Obj fun:", obj) }
        cat("  Ngroups:", n_groups ,"Nchanged:",sum(gamma_k_change, na.rm = TRUE),"\n")
        image(estimate, col=grey.colors(255)) # view estimate
      }
    } # end while (will break when no gammas change in a cycle of descent & fusion)

    ###############.
    ## Smoothing
    estimate_list[[i]] <- estimate # record the estimate
    i <- i + 1
    lam2 <- lam2 + delta

    if (length(unique(as.vector(groups))) == 1) { break } ## stop iterating if one group
  }

  return(estimate_list)
}

###-----------------------------------------------------###
###---- Implementation                                 ----
```

```
###-------------------------------------------------------###
system.time({

all_noisy_images_c <- list(cross_noi_s10, cross_noi_s40)
estimates.cross <- lapply(all_noisy_images_c, FUN = fused_lasso,
                          lam1=0, lam2=0, lam2_max=0.1, delta=0.02, verbose=TRUE)

save(estimates.cross, cross, all_noisy_images_c, file="cross_estimates.RData")

})
###-------------------------------------------------------###
###---- Evaulation                                       ----
###-------------------------------------------------------###

## Peak Signal-to-Noise Ratio
psnr <- function(y_true, y_est) {
  10*log(base = 10, x = ( 1/mean((y_true-y_est)^2) ) )
}
save(psnr, file="psnr.RData")

image(estimates.cross[[1]][[5]], col=grey.colors(255))
image(estimates.cross[[2]][[5]], col=grey.colors(255))
image(cross_noi_s10, col=grey.colors(255))

psnr_s10 <- sapply(estimates.cross[[1]], FUN = psnr, y_true=cross)
psnr_s40 <- sapply(estimates.cross[[2]], FUN = psnr, y_true=cross)


###-------------------------------------------------------###
###---- Implementation  (in flsa package)                ----
###-------------------------------------------------------###
{
library(flsa)

lambda2_seq <- seq(0,0.1,length=5)

# tmp <- flsa(all_noisy_images$house$s20, lambda1=0, lambda2=lambda2_seq, verbose=TRUE)

system.time({
  estimates.all <- lapply(all_noisy_images, function(x) {
    estimates.x <- lapply(x, FUN = flsa,
                          lambda1=0, lambda2=lambda2_seq, verbose=TRUE)
    return(estimates.x)
  })
})
#    user  system elapsed
# 1262.53    1.47 1293.36
# about 20 minutes for 9 256x256 images

save(estimates.all, all_noisy_images, all_true_images, file="all_estimates.RData")

}
###-------------------------------------------------------####

image(estimates.all$Cameraman256$s20[1,,], col=grey.colors(255))
```

References

Dabov, K., A. Foi, V. Katkovnik, and K. O. Egiazarian. 2007. "Image Denoising by Sparse 3-d Transform-Domain Collaborative Filtering." *IEEE Transactions on Image Processing* 16 (8): 2080–95. http://dl.acm.org/citation.cfm?id=2319074.2321415.

Friedman, J. H., T. Hastie, H. Hofling, and R. Tibshirani. 2007. "Pathwise Coordinate Optimization." *The Annals of Applied Statistics* 1 (2): 302–32. https://projecteuclid.org/euclid.aoas/1196438020.

Tibshirani, R., and P. Wang. 2008. "Spatial Smoothing and Hot Spot Detection for Cgh Data Using the Fused Lasso." *Biostatistics* 9 (1): 18–29. http://statweb.stanford.edu/.

Tibshirani, R., M. A. Saunders, S. Rosset, J. Zhu, and K. Knight. 2005. "Sparsity and Smoothness via the Fused Lasso." *Journal of the Royal Statistical Society Series B-Statistical Methodology* 67 (1): 91–108. http://onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2005.00490.x/abstract.

Xin, B., Y. Kawahara, Y. Wang, L. Hu, and W. Gao. 2016. "Efficient Generalized Fused Lasso and Its Applications." *ACM Transactions on Intelligent Systems and Technology* 7: 4. http://dl.acm.org/citation.cfm?id=2847421.