SDS 385: Homework 8

G. Paulon

November 7, 2016

Problem 1. Laplacian smoothing

(A) *Consider the Laplacian smoothing problem:*

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \frac{1}{2}||\boldsymbol{y}-\boldsymbol{x}||_2^2 + \frac{\lambda}{2}\boldsymbol{x}^T L\boldsymbol{x}$$

where L is the graph Laplacian. Show that the Laplacian matrix has the alternate representation $L = D^T D$, and therefore that penalty term $\boldsymbol{x}^T L\boldsymbol{x}$ can be rewritten as $\boldsymbol{x}^T L\boldsymbol{x} = ||D\boldsymbol{x}||_2^2$.

Since $L = D^T D$, then the penalty term can be rewritten as

$$\begin{aligned}\boldsymbol{x}^T L\boldsymbol{x} &= \boldsymbol{x}^T D^T D\boldsymbol{x}\\ &= (D\boldsymbol{x})^T(D\boldsymbol{x})\\ &= ||D\boldsymbol{x}||_2^2\end{aligned}$$

(B) *Show that the solution $\hat{\boldsymbol{x}}$ of the Laplacian-smoothing objective solves a linear system*

$$C\hat{\boldsymbol{x}} = \boldsymbol{b},$$

where C and $\boldsymbol{b}$ are matrices you name.

The minimization problem is

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} f(\boldsymbol{x}) = \min_{\boldsymbol{x}\in\mathbb{R}^n} \frac{1}{2}||\boldsymbol{y}-\boldsymbol{x}||_2^2 + \frac{\lambda}{2}\boldsymbol{x}^T L\boldsymbol{x}$$

which is easy to solve by taking the gradient with respect to $\boldsymbol{x}$ because the objective function is a quadratic form. In fact

$$\nabla f(\boldsymbol{x}) = -(\boldsymbol{y}-\hat{\boldsymbol{x}}) + \lambda L\hat{\boldsymbol{x}} = 0.$$

Therefore

$$\hat{\boldsymbol{x}} = (I+\lambda L)^{-1}\boldsymbol{y} = H\boldsymbol{y},$$

which is a linear smoother of $\boldsymbol{y}$.

(C) *Obtain the data "fmri-z.csv" from the class website. This is a 128x128 cross-sectional slice of data from an fMRI experiment on spatial memory. The data matrix reflects the underlying grid structure of the graph, i.e. entry (i,j) in the matrix corresponds to site (i,j) on the grid. Now consider the following four algorithms for solving the above linear system:*

1. *a direct solver that uses a sparse matrix factorization (e.g. sparse Cholesky or sparse LU)*
2. *the Gauss-Seidel method*
3. *the Jacobi iterative method*
4. *the conjugate gradient method*

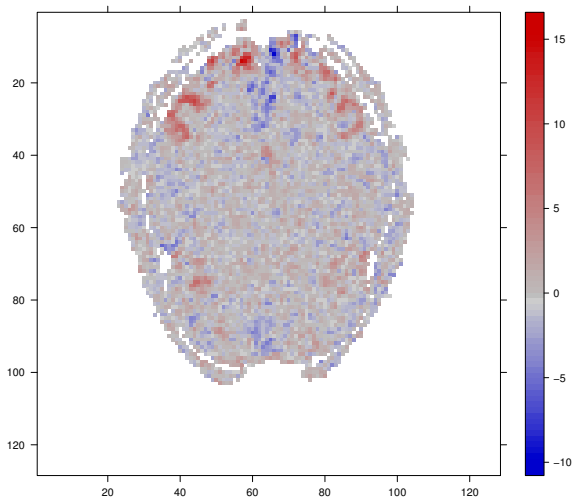


Figure 1: Original fMRI data.

You obviously known about the direct solver, but read up on the other three (all of which are iterative methods). Pick the direct solver (method 1) and any one of the other three methods - whichever you think makes the most sense for the problem, in light of having read about them all.

The data is illustrated in Figure 1. They consist of a grid of 128×128 pixels and, as one can see, they contain a lot of noise.

Therefore, the number of vertices of the graph is $n = |\mathcal{S}| = n_X \times n_Y = 128 \cdot 128 = 16384$. The notion of neighbourhood is defined as follows: two cells of the matrix are neighbours if one is on top, at the bottom, on the left or on the right of the other. Therefore, the diagonal cells are not neighbours. The number of edges, thus, is

$$m = |\mathcal{E}| = n_Y(n_X - 1) + n_X(n_Y - 1) = 2n_X n_Y - n_X - n_Y.$$

In fact, for each of the n_Y rows of the matrix there are $n_X - 1$ horizontal edges. Moreover, for each of the n_X columns of the matrix there are $n_Y - 1$ vertical edges. In our case

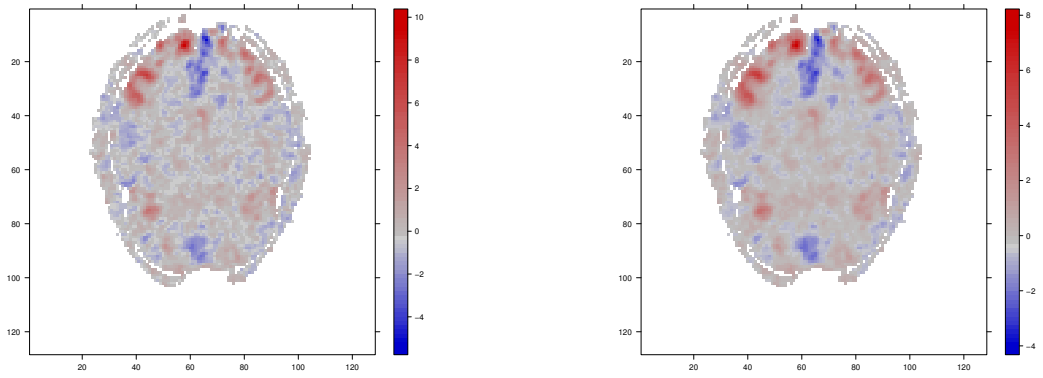
In the following, both the direct method and the Jacobi iterative solver have been implemented. The code is displayed in Listing A.1 and in Listing A.2. The direct solver takes advantage of R's builtin routines of the `Matrix` package, which exploits a sparse Cholesky decomposition.

The **Jacobi solver**, instead, is an iterative method that uses a decomposition of the matrix $I - \lambda L = D + R$, where D is a diagonal matrix and R contains only the extradiagonal elements of $I - \lambda R$. The solution is then obtained iteratively via

$$\boldsymbol{x}^{(t+1)} = D^{-1}(\boldsymbol{y} - R\boldsymbol{x}^{(t)}).$$

The inversion of a diagonal matrix is way less cumbersome than its dense counterpart, since it is sufficient to take the inverse of the scalar elements on the diagonal.

Both the direct and the iterative method (whose stopping criterion is the convergence of the parameter of interest) give the same result for different values of lambda. In Figure 2 we display the results for increasing levels of the penalty term λ. As one can see, the more λ is increased, the more smooth and blurred the figure looks like and the more information is lost.



(a) $\lambda = 1$ *(b)* $\lambda = 2$

Figure 2: Results of the Laplacian smoothing.

Problem 2. Graph fused lasso

Now consider a version of the spatial smoothing problem where we change the $\mathcal{L}_2$ *penalty to an* $\mathcal{L}_1$ *penalty:*

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \frac{1}{2}||\boldsymbol{y}-\boldsymbol{x}||_2^2 + \lambda||D\boldsymbol{x}||_1$$

where we recall that D *is the oriented edge matrix of the graph defined earlier. The penalty term rewards the solution for having small absolute first differences across the edges in the graph. This is called the graph fused lasso, or graph-based total-variation denoising.*

Since this does not have a closed-form solution, we will solve this problem via ADMM. There is a fairly obvious ADMM approach, based on rewriting the problem as

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \frac{1}{2}||\boldsymbol{y}-\boldsymbol{x}||_2^2 + \lambda||\boldsymbol{r}||_1$$
$$\text{subject to } D\boldsymbol{x} = \boldsymbol{r}.$$

The augmented Lagrangian for this problem is

$$L_\rho(x,r,v) = \frac{1}{2}||y-x||_2^2 + v^T(Dx-r) + \frac{\rho}{2}||Dx-r||_2^2 + \lambda||r||_1.$$

The x-update is then

$$\begin{aligned} x^{(k+1)} &= \underset{x}{\operatorname{argmin}}\, L_r ho(x, r(k), v^{(k)}) \\ &= \underset{x}{\operatorname{argmin}}\, \frac{1}{2}||y - x||_2^2 + v^T(Dx - r) + \frac{\rho}{2}||Dx - r||_2^2 \\ &\Rightarrow (I + \rho D^T D)x^{(k+1)} = y - D^T v^{(k)} + \rho D^T r = y - \rho D^T(r^{(k)} - u^{(k)}), \end{aligned}$$

where $u = v/\rho$.

The r-update is

$$\begin{aligned} r^{(k+1)} &= \underset{r}{\operatorname{argmin}}\, L_r ho(x^{(k+1)}, r, v^{(k)}) \\ &= \underset{r}{\operatorname{argmin}}\, \lambda||r||_1 + (v^{(k)})^T(Dx^{(k+1)} - r) + \frac{\rho}{2}||Dx - r||_2^2 \\ &= \underset{r}{\operatorname{argmin}}\, \frac{\lambda}{\rho}||r||_1 + (u^{(k)})^T(Dx^{(k+1)} - r) + \frac{1}{2}||Dx - r||_2^2 \\ &= \mathcal{S}_{\lambda/\rho}(Dx^{(k+1)} + u^{(k)}). \end{aligned}$$

The u-update is simply

$$u^{(k+1)} = u^{(k)} + Dx^{(k+1)} - r^{(k+1)}.$$

The primal residual, which checks how close we are to the primal solution, is

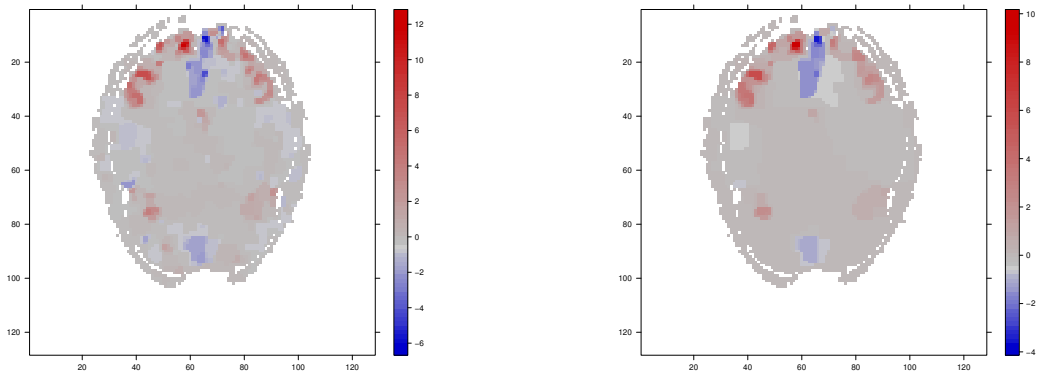
$$q^{(k+1)} = Dx^{(k+1)} - r^{(k+1)}$$

and the dual residual is

$$s^{(k+1)} = -\rho D^T(u^{(k+1)} - u^{(k)}).$$

As a convergence criterion, we use

$$\begin{cases} ||q^{(k+1)}||_2 \leq \sqrt{m}\varepsilon^{abs} + \varepsilon^{rel} \max\{||Dx^{(k+1)}||_2; ||r^{(k+1)}||_2\} = \varepsilon^{primal} \\ ||s^{(k+1)}||_2 \leq \sqrt{n}\varepsilon^{abs} + \varepsilon^{rel}||\rho D^T u^{(k+1)}||_2 = \varepsilon^{dual} \end{cases}$$



(a) $\lambda = 1$ *(b)* $\lambda = 2$

Figure 3: Results of the Graph fused lasso.

Appendix A

R code

```r
objective.l2 <- function(x, z, y, L, lambda){
  # Computes the objective function with L2 penalty
  # -----------------------------------------------
  # Args:
  #   - x: actual point
  #   - z: actual point
  #   - y: response vector of the data
  #   - L: laplacian penalty matrix
  #   - lambda: penalization parameter
  # Returns:
  #   - obj: the scalar associated with the objective function at beta
  # ----------------------------------------------------------------
  obj <- (1/2)*crossprod(x - y) + (lambda/2) * crossprod(z, L %*% z)
  return(obj)
}


laplacian.direct <- function(y, L, lambda){
  # Direct solver for the laplacian smoothing
  # -----------------------------------------
  # Args:
  #   - y: response vector of the observations (length n)
  #   - L: laplacian matrix
  #   - lambda: penalization parameter for the laplacian (threshold)
  # Returns:
  #   - x: values of the mean functional on the grid
  # ----------------------------------------------
  # Compute the matrix H of the linear system Hx = y
  H <- lambda * L
  diag(H) <- diag(H) + 1

  # Solve the linear system
  x <- solve(H, y)
  # R = chol(H)
  # u = forwardsolve(t(R), y)
  # x = backsolve(R, u)
  return (x)
}
```

Listing A.1: Laplacian smoothing implemented via direct method.

```r
laplacian.jacobi <- function(y, L, lambda, maxiter){
  # Jacobi iterative solver for the laplacian smoothing
  # ---------------------------------------------------
  # Args:
  #   - y: response vector of the observations (length n)
  #   - L: laplacian matrix
  #   - lambda: penalization parameter for the laplacian (threshold)
  # Returns:
  #   - x: values of the mean functional on the grid
  # ---------------------------------------------------
  # Compute the matrix H of the linear system Hx = y
  H <- lambda * L
  diag(H) <- diag(H) + 1

  # Solve the linear system
  x <- rep(0, dim(H)[1])
  D <- diag(H)
  R <- H
  diag(R) <- rep(0, dim(H)[1])

  for (iter in 1:maxiter){
    xold <- x
    x <- (1/D)*(y - R %*% xold)
    if (max(abs(x - xold)) < 1E-3){
      break;
    }
    else if (max(abs(x - xold)) >= 1E-3 & iter == maxiter){
      cat("WARNING: Jacobi Method did not converge\n")
      break;
    }
  }
  return (x)
}
```

Listing A.2: Laplacian smoothing implemented via Jacobi method.

```r
soft.thresholding <- function(x, lambda){
  # Computes the soft thresholding estimator
  # ----------------------------------------
  # Args:
  #   - x: vector of the observations
  #   - lambda: penalization parameter (threshold)
  # Returns:
  #   - theta: the soft thresholding estimator
  # ----------------------------------------
  theta <- sign(x) * pmax(rep(0, length(x)), abs(x) - lambda)
  return (theta)
}


objective.l1 <- function(x, z, y, L, lambda){
  # Computes the objective function with L1 penalty
  # -----------------------------------------------
  # Args:
  #   - x: actual point
  #   - z: actual point
```

```
  #   - y: response vector of the data
  #   - L: laplacian penalty matrix
  #   - lambda: penalization parameter
  # Returns:
  #   - obj: the scalar associated with the objective function at beta
  # ------------------------------------------------------------------
  obj <- (1/2)*crossprod(x - y) + (lambda/2) * crossprod(z)
  return(obj)
}


fusedlasso.refined <- function(y, D, x0, lambda, rho = 1, maxiter = 5000, tol.abs = 1E-5, tol.rel
    = 1E-5){
  # Optimized ADMM that allows to compute the Fused Lasso
  # -----------------------------------------------------
  # Args:
  #   - y: response vector of the observations (length n)
  #   - D: oriented edge matrix of the graph
  #   - x0: initial values of x
  #   - lambda: penalization parameter for the lasso (threshold)
  #   - rho: step size
  #   - maxiter: maximum number of iterations
  #   - tol.abs: tolerance defining convergence
  #   - tol.rel: tolerance defining convergence
  # Returns:
  #   - obj: values of the objective function for each iteration
  #   - x: values of the mean functional on the grid
  # ------------------------------------------------

  L <- crossprod(D)

  # Check if the inputs have the correct sizes
  if ((length(y) != dim(L)[1]) || (length(x0) != dim(L)[1])){
    stop("Incorrect input dimensions.")
  }

  # Rescaling the threshold to make it comparable to glmnet
  # lambda <- lambda * length(y)
  n <- length(x0)
  m <- dim(D)[1]

  # Initialize the data structures
  x <- array(NA, dim = n)
  x <- x0 # Initial guess
  z <- array(NA, dim = n)
  z <- rep(0, n)
  u <- array(NA, dim = n)
  u <- rep(0, n)
  t <- array(NA, dim = m)
  t <- rep(0, m)
  s <- array(NA, dim = m)
  s <- rep(0, m)
  obj <- array(NA, dim = maxiter)
  obj[1] <- objective.ll(z, z, y, L, lambda)

  # Precaching operations
  E <- L
```

```
  diag(E) <- diag(E) + 1

  for (iter in 2:maxiter){

    zold <- z

    # x - minimization step
    x <- (as.vector(y) + rho * (z - u)) / (1 + rho)

    # r - minimization step
    r <- soft.thresholding(as.vector(s - t), lambda)

    # (z, s) - minimization step
    Dtv <- crossprod(D, r + t)
    z <- solve(E, x + u + Dtv)
    s <- D %*% z

    # (u, t) - minimization step
    res <- x - z
    u <- u + res
    t <- t + r - s

    # Compute log-likelihood
    obj[iter] <- objective.l1(z, z, y, L, lambda)

    rnorm <- sqrt(sum(res^2))
    snorm <- rho * sqrt(sum((z - zold)^2))
    eprim <- sqrt(n) * tol.abs + tol.rel * max(c(sqrt(sum(x^2)), sqrt(sum(z^2)), 0))
    edual <- sqrt(n) * tol.abs + tol.rel * rho * sqrt(sum(u^2))
    # Convergence check
    if ( (rnorm < eprim) & (snorm < edual) ){
      cat('Algorithm has converged after', iter, 'iterations')
      obj <- obj[1:iter]
      break;
    }
    else if ( (iter == maxiter) & ((rnorm >= eprim) || (snorm >= edual)) ){
      print('WARNING: algorithm has not converged')
      break;
    }
  }
  return(list("obj" = obj, "x" = z))
}
```

Listing A.3: Fused Lasso algorithm.