

Regularized Precision Matrix Estimation via ADMM

Matt Galloway

March 3, 2018

Abstract

ADMMsigma is an R package that estimates a penalized precision matrix via the alternating direction method of multipliers (ADMM) algorithm. This report will provide a brief overview of the algorithm and detail how it can be utilized to estimate precision matrices of joint normal distributions. In addition, examples and simulation results will be provided for **ADMMsigma**.

Contents

1	Introduction	1
2	Regularized Precision Matrix Estimation	2
2.1	Condensed-Form ADMM	3
2.2	Algorithm	3
3	R Package	6
3.1	Installation	6
3.2	Usage	6
3.3	Benchmark	10

1 Introduction

Suppose we want to minimize $f(x) + g(z)$ subject to the constraint that $Ax + Bz = c$. For now, we will take $x \in \mathbb{R}^n, z \in \mathbb{R}^m, A \in \mathbb{R}^{p \times n}, B \in \mathbb{R}^{p \times m}, c \in \mathbb{R}^p$ – though we will later consider cases where x and z are matrices. The *augmented lagrangian* is constructed as follows:

$$L_\rho(x, z, y) = f(x) + g(z) + y^T(Ax + Bz - c) + \frac{\rho}{2} \|Ax + Bz - c\|_2^2$$

where $y \in \mathbb{R}^p$ is the lagrange multiplier. The optimal value is

$$p^* = \inf \{f(x) + g(z) | Ax + Bz = c\}$$

Clearly, the minimization problem under the augmented lagrangian (RE-WORK) is equivalent to that of the usual lagrangian since any feasible point (x, z) satisfies the constraint $\rho \|Ax + Bz - c\|_2^2 / 2 = 0$.

The ADMM algorithm consists of the following repeated iterations:

$$x^{k+1} := \arg \min_x L_\rho(x, z^k, y^k) \tag{1}$$

$$z^{k+1} := \arg \min_z L_\rho(x^{k+1}, z, y^k) \tag{2}$$

$$y^{k+1} := y^k + \rho(Ax^{k+1} + Bz^{k+1} - c) \tag{3}$$

A more complete introduction to the algorithm – specifically how it arose out of *dual ascent* and *method of multipliers* – can be found in Boyd, et al. (2011).

2 Regularized Precision Matrix Estimation

We now consider the case where X_1, \dots, X_n are iid $N_p(\mu, \Sigma)$ and we are tasked with estimating the precision matrix, denoted $\Omega \equiv \Sigma^{-1}$. The maximum likelihood estimator for Ω is

$$\hat{\Omega} = \arg \min_{\Omega \in S_+^p} \{Tr(S\Omega) - \log \det(\Omega)\}$$

where $S = \sum_{i=1}^n (X_i - \bar{X})(X_i - \bar{X})^T / n$. It is straight forward to show that when the solution exists, $\hat{\Omega} = S^{-1}$.

We can further construct a penalized likelihood estimator by adding a penalty term, $P_\lambda(\Omega)$, to the likelihood:

$$\hat{\Omega}_\lambda = \arg \min_{\Omega \in S_+^p} \{Tr(S\Omega) - \log \det(\Omega) + P_\lambda(\Omega)\}$$

Throughout the rest of this document we will take $P_\lambda(\Omega)$ to be $P_\lambda(\Omega) = \lambda \left[\frac{1-\alpha}{2} \|\Omega\|_F^2 + \alpha \|\Omega\|_1 \right]$ so that the full penalized likelihood is as follows:

$$\hat{\Omega}_\lambda = \arg \min_{\Omega \in S_+^p} \left\{ Tr(S\Omega) - \log \det(\Omega) + \lambda \left[\frac{1-\alpha}{2} \|\Omega\|_F^2 + \alpha \|\Omega\|_1 \right] \right\}$$

where $0 \leq \alpha \leq 1$, $\lambda > 0$, $0 < \eta < 2$, $\|\cdot\|_F^2$ is the Frobenius norm and we define $\|A\|_1 = \sum_{i,j} |A_{ij}|$. This penalty is closely related to the elastic-net penalty explored by Hui Zou and Trevor Hastie [4]. Clearly, when $\alpha = 0$ this reduces to a ridge-type penalty and when $\alpha = 1$ this reduces to a lasso-type penalty.

By letting f be equal to the non-penalized likelihood and g equal to $P_\lambda(\Omega)$, our goal is to minimize the full augmented lagrangian where the constraint is that $\Omega - Z$ is equal to zero:

$$L_\rho(\Omega, Z, \Lambda) = f(\Omega) + g(Z) + Tr[\Lambda(\Omega - Z)] + \frac{\rho}{2} \|\Omega - Z\|_F^2$$

The ADMM algorithm for regularized precision matrix estimation is

$$\Omega^{k+1} = \arg \min_{S\Omega} \left\{ Tr(\Omega) - \log \det(\Omega) + Tr[\Lambda^k(\Omega - Z^k)] + \frac{\rho}{2} \|\Omega - Z^k\|_F^2 \right\} \quad (4)$$

$$Z^{k+1} = \arg \min_Z \left\{ \lambda \left[\frac{1-\alpha}{2} \|Z\|_F^2 + \alpha \|Z\|_1 \right] + Tr[\Lambda^k(\Omega^{k+1} - Z)] + \frac{\rho}{2} \|\Omega^{k+1} - Z\|_F^2 \right\} \quad (5)$$

$$\Lambda^{k+1} = \Lambda^k + \rho(\Omega^{k+1} - Z^{k+1}) \quad (6)$$

2.1 Condensed-Form ADMM

An alternate form of the ADMM algorithm can be constructed by scaling the dual variable. Let us define $R^k = \Omega - Z^k$ and $U^k = \Lambda^k / \rho$. Then

$$\begin{aligned} \text{Tr} [\Lambda^k (\Omega - Z^k)] + \frac{\rho}{2} \|\Omega - Z^k\|_F^2 &= \text{Tr} [\Lambda^k R^k] + \frac{\rho}{2} \|R^k\|_F^2 \\ &= \frac{\rho}{2} \|R^k + \Lambda^k / \rho\|_F^2 - \frac{\rho}{2} \|\Lambda^k / \rho\|_F^2 \\ &= \frac{\rho}{2} \|R^k + U^k\|_F^2 - \frac{\rho}{2} \|U^k\|_F^2 \end{aligned}$$

The condensed-form can now be written as follows:

$$\Omega^{k+1} = \arg \min_{\Omega} \left\{ \text{Tr}(\Omega) - \log \det(\Omega) + \frac{\rho}{2} \|\Omega - Z^k + U^k\|_F^2 \right\} \quad (7)$$

$$Z^{k+1} = \arg \min_Z \left\{ \lambda \left[\frac{1-\alpha}{2} \|Z\|_F^2 + \alpha \|Z\|_1 \right] + \frac{\rho}{2} \|\Omega^{k+1} - Z + U^k\|_F^2 \right\} \quad (8)$$

$$U^{k+1} = U^k + \Omega^{k+1} - Z^{k+1} \quad (9)$$

More generally (in vector form),

$$x^{k+1} := \arg \min_x \left\{ f(x) + \frac{\rho}{2} \|Ax + Bz^k - c + u^k\|_2^2 \right\} \quad (10)$$

$$z^{k+1} := \arg \min_z \left\{ g(z) + \frac{\rho}{2} \|Ax^{k+1} + Bz - c + u^k\|_2^2 \right\} \quad (11)$$

$$u^{k+1} := u^k + Ax^{k+1} + Bz^{k+1} - c \quad (12)$$

Note that there are limitations to using this method. For instance, because the dual variable is scaled by ρ (the step size), this form limits one to using a constant step size (without making further adjustments to U^k) – a limitation that could prolong the convergence rate.

2.2 Algorithm

$$\begin{aligned} \Omega^{k+1} &= \arg \min_{\Omega} \left\{ \text{Tr}(\Omega) - \log \det(\Omega) + \frac{\rho}{2} \|\Omega - Z^k + U^k\|_F^2 \right\} \\ Z^{k+1} &= \arg \min_Z \left\{ \lambda \left[\frac{1-\alpha}{2} \|Z\|_F^2 + \alpha \|Z\|_1 \right] + \frac{\rho}{2} \|\Omega^{k+1} - Z + U^k\|_F^2 \right\} \\ U^{k+1} &= U^k + \Omega^{k+1} - Z^{k+1} \end{aligned}$$

1. Decompose $S + \rho(U^k - Z^k) = VQV^T$.

$$\Omega^{k+1} = \frac{1}{2\rho} V \left[-Q + (Q^2 + 4\rho I_p)^{1/2} \right] V^T$$

2. Elementwise soft-thresholding for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

$$\begin{aligned} Z_{ij}^{k+1} &= \frac{1}{\lambda(1-\alpha) + \rho} \text{sign}(\Omega_{ij}^{k+1} + U_{ij}^k) (\rho |\Omega_{ij}^{k+1} + U_{ij}^k| - \lambda\eta\alpha)_+ \\ &= \frac{1}{\lambda(1-\alpha) + \rho} \text{Soft}(\rho(\Omega_{ij}^{k+1} + U_{ij}^k), \lambda\eta\alpha) \end{aligned}$$

3. Update U .

$$U^{k+1} = U^k + \Omega^{k+1} - Z^{k+1}$$

2.2.1 Proof of (1):

(Work in progress.)

Code snippet:

Note this is not the actual code. The real code is written in c++.

```
# ridge penalized precision matrix
# function
RIDGEsigma = function(S, lam) {

  # dimensions
  p = dim(S)[1]

  # gather eigen values of S (spectral
# decomposition)
  e.out = eigen(S, symmetric = TRUE)

  # augment eigen values for omega hat
  new.evs = (-e.out$val + sqrt(e.out$val^2 +
    4 * lam))/(2 * lam)

  # compute omega hat for lambda (zero
# gradient equation)
  omega = tcrossprod(e.out$vec * rep(new.evs,
    each = p), e.out$vec)

  # compute gradient
  grad = S - qr.solve(omega) + lam * omega

  return(list(omega = omega, gradient = grad))
}
```

2.2.2 Proof of (2)

(Work in progress.)

Code snippet:

Note this is not the actual code. The real code is written in c++.

```
# ADMMsigma function
ADMMsigma = function(X = NULL, S = NULL,
  lam, alpha = 1, rho = 2, mu = 10, tau1 = 2,
  tau2 = 2, tol1 = 1e-04, tol2 = 1e-04,
  maxit = 1000) {

  # compute sample covariance matrix, if
  # necessary
  if (is.null(S)) {

    # covariance matrix
    n = dim(X)[1]
    S = (n - 1)/n * cov(X)

  }

  # allocate memory
  p = dim(S)[1]
  criterion = TRUE
  iter = lik = s = r = eps1 = eps2 = 0
  new.Z = Y = Omega = matrix(0, nrow = p,
    ncol = p)

  # loop until convergence
  while (criterion && (iter <= maxit)) {

    # ridge equation (1) gather eigen values
    # (spectral decomposition)
    Z = new.Z
    Omega = sigma_ridge(S + Y - rho *
      Z, lam = rho)$omega

    # penalty equation (2) soft-thresholding
    new.Z = soft(Y + rho * Omega, lam *
      alpha)/(lam * (1 - alpha) + rho)

    # update U (3)
    Y = Y + rho * (Omega - new.Z)

    # calculate new rho
    s = sqrt(sum((rho * (new.Z - Z))^2))
    r = sqrt(sum((Omega - new.Z)^2))
    rho = rho * (tau1 * (r > mu * s) +
      (s > mu * r)/tau2 + (s/mu <=
```

```

        r & r <= mu * s))
    iter = iter + 1

    # stopping criterion
    eps1 = p * tol1 + tol2 * max(sqrt(sum(Omega^2)),
                                   sqrt(sum(new.Z^2)))
    eps2 = p * tol1 + tol2 * sqrt(sum(Y^2))
    criterion = (r >= eps1 || s >= eps2)

  }
  return(list(Iterations = iter, Omega = Omega))
}

```

3 R Package

3.1 Installation

```

# The easiest way to install is from the
# development version from GitHub:
# install.packages('devtools')
devtools::install_github("MGallow/ADMMsigma")

```

Important: if using operating systems other than Mac or Windows, you will have to download the package as source and add a `src/Makevars` file (identical to the `Makevars.win` already included).

If there are any issues/bugs, please let me know: [github](#). You can also contact me via my website. Pull requests are welcome!

3.2 Usage

```

library(ADMMsigma)

# generate data from a dense matrix for
# example first compute covariance matrix
S = matrix(0, nrow = 5, ncol = 5)

for (i in 1:5) {
  for (j in 1:5) {
    S[i, j] = 0.9^(i != j)
  }
}

```

```

}

# generate 100x5 matrix with rows drawn
# from iid  $N_p(0, S)$ 
Z = matrix(rnorm(100 * 5), nrow = 100, ncol = 5)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*%
         t(out$vectors)
X = Z %*% S.sqrt

# elastic-net type penalty (use CV for
# optimal lambda and alpha)
ADMMsigma(X)

##
## Iterations:
## [1] 35
##
## Tuning parameters:
##      log10(lam)  alpha
## [1,]      -2.5    0.6
##
## Omega:
##      [,1]      [,2]      [,3]      [,4]      [,5]
## [1,]  7.50103 -1.71813 -2.17055 -1.21055 -1.95217
## [2,] -1.71813  6.88957 -2.51084 -1.42759 -1.14920
## [3,] -2.17055 -2.51084  8.17508 -1.79449 -1.38145
## [4,] -1.21055 -1.42759 -1.79449  6.25581 -1.97081
## [5,] -1.95217 -1.14920 -1.38145 -1.97081  6.53472

# ridge penalty (use CV for optimal
# lambda)
ADMMsigma(X, alpha = 0)

##
## Iterations:
## [1] 39
##
## Tuning parameters:
##      log10(lam)  alpha
## [1,]      -3      0
##
## Omega:
##      [,1]      [,2]      [,3]      [,4]      [,5]
## [1,]  7.92134 -1.80920 -2.32991 -1.24122 -2.07704
## [2,] -1.80920  7.24953 -2.70397 -1.48210 -1.17634
## [3,] -2.32991 -2.70397  8.69320 -1.89807 -1.43455
## [4,] -1.24122 -1.48210 -1.89807  6.54511 -2.08445
## [5,] -2.07704 -1.17634 -1.43455 -2.08445  6.84977

# lasso penalty (lam = 0.1)
ADMMsigma(X, lam = 0.1, alpha = 1)

##

```

```

## Iterations:
## [1] 10
##
## Tuning parameters:
##      log10(lam)  alpha
## [1,]         -1      1
##
## Omega:
##      [,1]      [,2]      [,3]      [,4]      [,5]
## [1,]  2.80422 -0.62667 -0.67987 -0.55554 -0.66056
## [2,] -0.62667  2.66766 -0.75422 -0.60328 -0.54369
## [3,] -0.67987 -0.75422  2.88706 -0.65217 -0.57631
## [4,] -0.55554 -0.60328 -0.65217  2.53979 -0.69921
## [5,] -0.66056 -0.54369 -0.57631 -0.69921  2.60963

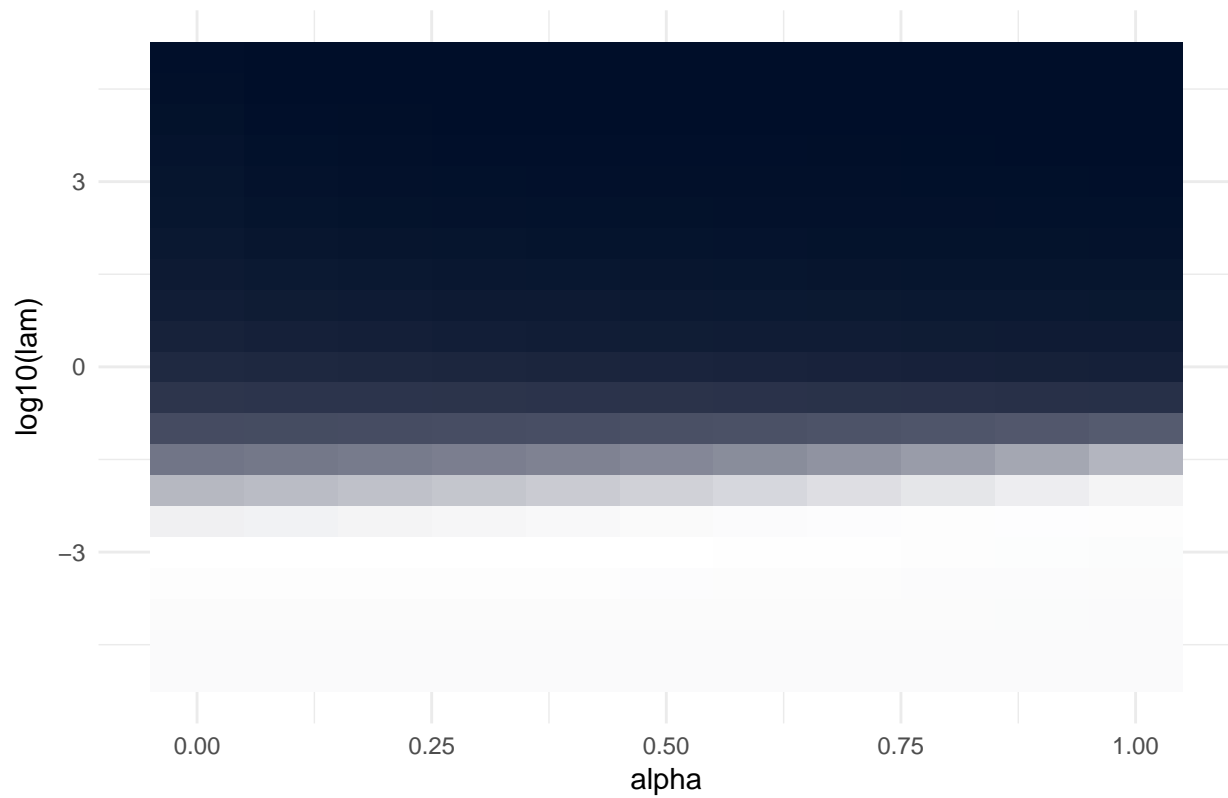
# ridge penalty no ADMM
RIDGEsigma(X, lam = 10^seq(-8, 8, 0.01))

##
## Tuning parameter:
##      lam  log10(alpha)
## [1,]  0.001          -3.1
##
## Omega:
##      [,1]      [,2]      [,3]      [,4]      [,5]
## [1,]  8.08365 -1.83697 -2.40822 -1.24051 -2.12903
## [2,] -1.83697  7.38256 -2.79786 -1.49474 -1.17783
## [3,] -2.40822 -2.79786  8.92397 -1.94165 -1.44583
## [4,] -1.24051 -1.49474 -1.94165  6.63854 -2.12649
## [5,] -2.12903 -1.17783 -1.44583 -2.12649  6.95522

# produce CV heat map for ADMMsigma
ADMMsigma(X) %>% plot

```


Heatmap of Cross-Validation Errors



```
# produce CV heat map for RIDGESigma
RIDGESigma(X, lam = 10seq(-8, 8, 0.01)) %>%
  plot
```

Heatmap of Cross-Validation Errors



3.3 Benchmark

3.3.1 Computer Specs:

- MacBook Pro (Late 2016)
- Processor: 2.9 GHz Intel Core i5
- Memory: 8GB 2133 MHz
- Graphics: Intel Iris Graphics 550

```
# generate data from tri-diagonal  
# (sparse) matrix for example first  
# compute covariance matrix (can confirm  
# inverse is tri-diagonal)  
S = matrix(0, nrow = 100, ncol = 100)  
  
for (i in 1:100) {  
  for (j in 1:100) {  
    S[i, j] = 0.7^(abs(i - j))  
  }  
}  
  
# generate 1000x100 matrix with rows  
# drawn from iid  $N_p(0, S)$   
Z = matrix(rnorm(1000 * 100), nrow = 1000,
```

```

ncol = 100)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %>% diag(out$values^0.5) %>%
  t(out$vectors)
X = Z %>% S.sqrt

# glasso
microbenchmark(glasso(s = S, rho = 0.1),
  times = 5)

## Unit: milliseconds
##           expr      min      lq      mean     median      uq
## glasso(s = S, rho = 0.1) 51.43172 52.78428 57.35826 54.42483 62.32629
##           max neval
## 65.82417      5

# benchmark ADMMsigma - default tolerance
microbenchmark(ADMMsigma(S = S, lam = 0.1,
  alpha = 1, tol1 = 1e-04, tol2 = 1e-04))

## Unit: milliseconds
##                                     expr
## ADMMsigma(S = S, lam = 0.1, alpha = 1, tol1 = 1e-04, tol2 = 1e-04)
##           min      lq      mean     median      uq      max neval
## 60.77063 61.85681 68.30814 64.82911 70.10486 135.8923 100

# benchmark ADMMsigma - tolerance 1e-8
microbenchmark(ADMMsigma(S = S, lam = 0.1,
  alpha = 1, tol1 = 1e-08, tol2 = 1e-08))

## Unit: milliseconds
##                                     expr
## ADMMsigma(S = S, lam = 0.1, alpha = 1, tol1 = 1e-08, tol2 = 1e-08)
##           min      lq      mean     median      uq      max neval
## 274.1254 275.8825 279.9305 277.6501 280.2129 311.3568 100

# benchmark ADMMsigma CV - likelihood
# convergence criteria
microbenchmark(ADMMsigma(X, lam = 10^seq(-8,
  8, 0.1), alpha = 1, crit = "loglik"),
  times = 5)

## Unit: seconds
##                                     expr
## ADMMsigma(X, lam = 10^seq(-8, 8, 0.1), alpha = 1, crit = "loglik")
##           min      lq      mean     median      uq      max neval
## 23.73195 23.95511 24.06034 24.05933 24.2458 24.30953 5

# benchmark ADMMsigma CV
microbenchmark(ADMMsigma(X, lam = 10^seq(-8,
  8, 0.1), alpha = 1), times = 5)

## Unit: seconds
##                                     expr      min      lq
## ADMMsigma(X, lam = 10^seq(-8, 8, 0.1), alpha = 1) 13.28752 13.36715
##           mean     median      uq      max neval

```

13.54836 13.5563 13.70937 13.82146 5

References

- [1] Boyd, Stephen, et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers." *Foundations and Trends® in Machine Learning* 3.1 (2011): 1-122.
- [2] Polson, Nicholas G., James G. Scott, and Brandon T. Willard. "Proximal algorithms in statistics and machine learning." *Statistical Science* 30.4 (2015): 559-581.
- [3] Marjanovic, Goran, and Victor Solo. "On l_q optimization and matrix completion." *IEEE Transactions on signal processing* 60.11 (2012): 5714-5724.
- [4] Zou, Hui, and Trevor Hastie. "Regularization and variable selection via the elastic net." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2 (2005): 301-320.