Regularized Precision Matrix Estimation via ADMM

Matt Galloway

February 27, 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

Suppose we want to minimize $f(x) + g(z)$ subject to the constraint that $Ax + Bz = c$. For now, we will take $x \in \mathbb{R}^n, z \in \mathbb{R}^m, A \in \mathbb{R}^{p \times m}, B \in \mathbb{R}^{p \times m}, c \in \mathbb{R}^p$ – 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:

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

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

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

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 $X_1, ..., X_n$ 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

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

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, $P_\lambda\left(\Omega\right)$, to the likelihood:

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

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

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

where $0 \leq \alpha \leq 1$, $\lambda > 0$, $0 < \eta < 2$, $\left\|\cdot\right\|_F^2$ is the Frobenius norm and we define $\left\|A\right\|_1 = \sum_{i,j} |A_{ij}|$. 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 $P_\lambda\left(\Omega\right)$, our goal is to minimize the full augmented lagrangian where the constraint is that $\Omega - Z$ is equal to zero:

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

The ADMM algorithm for regularized precision matrix estimation is

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

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

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

2.1 Condensed-Form ADMM

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

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

The condensed-form can now be written as follows:

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

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

$$U^{k+1} = U^k + \Omega^{k+1} - Z^{k+1} \tag{9}$$

More generally (in vector form),

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

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

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

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

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

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

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

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

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

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
sigma_ridge = 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")
```

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 tri-diagonal
# (sparse) matrix for example first
# compute covariance matrix (can confirm
# inverse is tri-diagonal)
S = matrix(0, nrow = 5, ncol = 5)

for (i in 1:5) {
    for (j in 1:5) {
        S[i, j] = 0.7^(abs(i - j))
    }
```

```
}

# generate 100x5 matrix with rows drawn
# from iid N_p(0, S)
Z = matrix(rnorm(100 * 10), 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


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

```
## $Iterations
## [1] 23
##
## $Parameters
##              lam alpha
## [1,] 0.003162278     0
##
## $Omega
##            [,1]        [,2]        [,3]        [,4]        [,5]
## [1,]  2.0746940 -1.30815999 -0.12950418  0.14630512 -0.12711236
## [2,] -1.3081600  2.82710268 -1.61336554 -0.01480074  0.11456474
## [3,] -0.1295042 -1.61336554  3.35434124 -1.40662152  0.03177144
## [4,]  0.1463051 -0.01480074 -1.40662152  2.53748507 -1.34611340
## [5,] -0.1271124  0.11456474  0.03177144 -1.34611340  1.78436934
##
## $Gradient
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,] -1.613060e-05  4.479649e-05 -5.353349e-05  2.647107e-05 -6.939976e-06
## [2,]  4.479649e-05 -1.387507e-04  1.788964e-04 -9.847227e-05  3.103029e-05
## [3,] -5.353349e-05  1.788964e-04 -2.413587e-04  1.404451e-04 -4.780782e-05
## [4,]  2.647107e-05 -9.847227e-05  1.404451e-04 -8.687203e-05  3.184993e-05
## [5,] -6.939976e-06  3.103029e-05 -4.780782e-05  3.184993e-05 -1.262593e-05
```

```
# lasso penalty (use CV for optimal
# lambda)
ADMMsigma(X)
```

```
## $Iterations
## [1] 29
##
## $Parameters
##             lam alpha
## [1,] 0.03162278     1
##
## $Omega
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,]  1.809005e+00 -1.0510166429 -0.1107362954  9.442773e-05 -0.0139966216
## [2,] -1.051017e+00  2.3399469916 -1.2611800834  4.398794e-04  0.0004788895
## [3,] -1.107363e-01 -1.2611800834  2.7161818402 -1.095457e+00  0.0001361328
```

```
## [4,]  9.442773e-05  0.0004398794 -1.0954569475  2.116030e+00 -1.0960606468
## [5,] -1.399662e-02  0.0004788895  0.0001361328 -1.096061e+00  1.5602227569
##
## $Gradient
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,] -2.783800e-05  5.457323e-05 -2.093457e-05  3.796755e-02 -3.719617e-05
## [2,]  5.457323e-05 -1.099099e-04  2.279745e-05  5.225614e-02  4.112619e-02
## [3,] -2.093457e-05  2.279745e-05 -1.232389e-06 -3.178364e-05  6.212127e-02
## [4,]  3.796755e-02  5.225614e-02 -3.178364e-05 -5.129313e-06 -5.054219e-07
## [5,] -3.719617e-05  4.112619e-02  6.212127e-02 -5.054219e-07 -1.050261e-05
```

```
# lasso penalty (lam = 0.1)
ADMMsigma(X, lam = 0.1)
```

```
## $Iterations
## [1] 17
##
## $Parameters
##      lam alpha
## [1,] 0.1     1
##
## $Omega
##               [,1]          [,2]        [,3]          [,4]          [,5]
## [1,]  1.3918423320 -0.6634654821 -0.14963980 -0.0000926349 -0.0107642088
## [2,] -0.6634654821  1.6027504106 -0.73849014 -0.0814963350 -0.0003599139
## [3,] -0.1496398039 -0.7384901374  1.83072039 -0.6578393666 -0.0724977476
## [4,] -0.0000926349 -0.0814963350 -0.65783937  1.4804623558 -0.7207809687
## [5,] -0.0107642088 -0.0003599139 -0.07249775 -0.7207809687  1.2070059505
##
## $Gradient
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,] -2.149774e-05  5.114656e-05 -4.727341e-05 -1.307474e-02  2.637115e-05
## [2,]  5.114656e-05 -1.316698e-04  1.559476e-04 -4.250951e-05 -1.389460e-02
## [3,] -4.727341e-05  1.559476e-04 -2.119649e-04  1.123417e-04 -2.460791e-05
## [4,] -1.307474e-02 -4.250951e-05  1.123417e-04 -9.113770e-05  4.101866e-05
## [5,]  2.637115e-05 -1.389460e-02 -2.460791e-05  4.101866e-05 -1.893766e-05
```

```
# elastic-net type penalty (use CV for
# optimal lambda)
ADMMsigma(X, alpha = 0.5)
```

```
## $Iterations
## [1] 22
##
## $Parameters
##       lam alpha
## [1,] 0.01   0.5
##
## $Omega
##             [,1]        [,2]          [,3]        [,4]          [,5]
## [1,]  1.98299722 -1.20777019 -0.1295551129  0.07125308 -0.0679080210
## [2,] -1.20777019  2.63117898 -1.4678431224  0.00071707  0.0583052873
## [3,] -0.12955511 -1.46784312  3.1005894467 -1.27088115  0.0002294593
## [4,]  0.07125308  0.00071707 -1.2708811509  2.36502155 -1.2452965205
## [5,] -0.06790802  0.05830529  0.0002294593 -1.24529652  1.7085068149
```

```
##
## $Gradient
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,] -1.690816e-05  4.687992e-05 -3.432799e-05 -1.486839e-05  1.624189e-05
## [2,]  4.687992e-05 -1.297548e-04  1.075292e-04  9.406325e-03 -4.974490e-05
## [3,] -3.432799e-05  1.075292e-04 -1.263860e-04  5.558806e-06  6.196174e-03
## [4,] -1.486839e-05  9.406325e-03  5.558806e-06 -1.377996e-05 -4.674862e-06
## [5,]  1.624189e-05 -4.974490e-05  6.196174e-03 -4.674862e-06 -1.614829e-06
```

```
# elastic-net type penalty (use CV for
# optimal lambda and alpha)
ADMMsigma(X, lam = 10^seq(-8, 8, 0.1), alpha = seq(0,
    1, 0.1))
```

```
## $Iterations
## [1] 30
##
## $Parameters
##             lam alpha
## [1,] 0.01584893     1
##
## $Omega
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,]  1.9531412835 -1.1998703764 -8.617752e-02  0.0008293242 -5.766522e-03
## [2,] -1.1998703764  2.6281730520 -1.449680e+00  0.0001652174  1.373400e-02
## [3,] -0.0861775231 -1.4496799860  3.024246e+00 -1.2225360721 -2.206938e-05
## [4,]  0.0008293242  0.0001652174 -1.222536e+00  2.3331396748 -1.219847e+00
## [5,] -0.0057665219  0.0137340033 -2.206938e-05 -1.2198468637  1.681425e+00
##
## $Gradient
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,] -6.271016e-05  1.286304e-04 -8.867450e-05 -1.779386e-05  3.012615e-05
## [2,]  1.286304e-04 -2.630406e-04  1.908901e-04  6.286280e-03 -4.258648e-05
## [3,] -8.867450e-05  1.908901e-04 -1.800293e-04  5.302693e-05 -9.232057e-03
## [4,] -1.779386e-05  6.286280e-03  5.302693e-05 -7.304554e-05  3.697555e-05
## [5,]  3.012615e-05 -4.258648e-05 -9.232057e-03  3.697555e-05 -2.207962e-05
```

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 = 10, ncol = 10)
```

```
for (i in 1:10) {
    for (j in 1:10) {
        S[i, j] = 0.7^(abs(i - j))
    }
}

# generate 1000x100 matrix with rows
# drawn from iid N_p(0, S)
Z = matrix(rnorm(100 * 10), nrow = 100, ncol = 10)
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))
```

```
## Unit: microseconds
##                       expr     min       lq     mean   median      uq
##  glasso(s = S, rho = 0.1) 259.369 291.1985 399.4992 318.4645 384.627
##       max neval
##  3048.748   100
```

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

```
## Unit: microseconds
##                                                    expr     min       lq
##  ADMMsigma(S = S, lam = 0.1, tol1 = 1e-04, tol2 = 1e-04) 566.707 683.0385
##      mean   median       uq      max neval
##  1498.455 1038.865 1485.661 11377.16   100
```

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

```
## Unit: milliseconds
##                                                    expr      min       lq
##  ADMMsigma(S = S, lam = 0.1, tol1 = 1e-08, tol2 = 1e-08) 1.586548 1.666134
##      mean   median       uq      max neval
##  1.895258 1.779635 2.005842 4.388559   100
```

```
# benchmark ADMMsigma CV - likelihood
# convergence criteria
microbenchmark(ADMMsigma(X, crit = "lik"))
```

```
## Unit: milliseconds
##                         expr      min       lq     mean   median       uq
##  ADMMsigma(X, crit = "lik") 24.56141 27.39162 31.52265 29.17019 32.19525
##       max neval
##  121.8315   100
```

```
# benchmark ADMMsigma CV
microbenchmark(ADMMsigma(X, lam = 10^seq(-8,
    8, 0.1), alpha = seq(0, 1, 0.1)))
```

```
## Unit: seconds
##                                                                 expr     min
##  ADMMsigma(X, lam = 10^seq(-8, 8, 0.1), alpha = seq(0, 1, 0.1)) 3.17915
##        lq    mean   median       uq      max neval
##  3.298076 3.46368 3.387187 3.508797 4.388642   100
```

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.