4 - Computing Least Squares Solutions

Computing a least squares solution involves many of the concepts covered in sections 2 and 3 such as matrix multiplication and matrix inverse. Thus there are efficient and inefficient ways to compute a least squares solution given the problem setup. More specifically, if the problem involves a psd matrix then one can take advantage of the cholesky decomposition methods which were shown to be advantageous.

4.1 - Ordinary Least Squares (OLS)

For solution S , data matrix X and solution space Y:

$$
\begin{aligned}
XS &= Y \\
X^T XS &= X^T Y \\
S &= (X^T X)^{-1} X^T Y
\end{aligned}
$$

One can make use of the pseudo inverse of X as its dimensions are arbitrary and may be singular:

$$
\begin{aligned}
XS &= Y \\
X^+ XS &= X^+ Y \\
S &\approx X^+ Y
\end{aligned}
$$

4.1.1 - SciPy OLS

To compute the ordinary least squares solution the pseudo inverse is used. This is done with the SciPy function `pinv()` from the `scipy.linalg` function base. More information on the `pinv()` function can be found on the SciPy documentation page [9].

```
X = np.random.uniform(size = (3,2))
Y = np.array([1,1,1])
X,Y
```

```
(array([[0.67459005, 0.33554957],
        [0.48042258, 0.86050783],
        [0.64218228, 0.429431  ]]),
 array([1, 1, 1]))
```

```
inv = la.pinv(X)
S = inv @ Y
X @ S # approximate solution to y
```

```
array([0.99860208, 0.99970045, 1.00169257])
```

4.1.2 - Python statsmodels Module

As per the statsmodels documentation page [11] statsmodels: Econometric and Statistical Modeling with Python "statsmodels is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration." As with the previous Python modules introduced, the tools we required can be imported with the call `import statsmodels.api as sm`. It should be noted that version 0.13.2 of the statsmodels module is being used.

```
import statsmodels.api as sm
print(sm.__version__)
```

```
0.13.2
```

Here the module is used to compute least square solutions. The functions `sm.OLS()` and `sm.GLS()` are used to compute the ordinary and generalized least square solutions.

4.1.3 - statsmodels OLS

Notice how the `OLS()` function produces the exact same result as the raw computation.

```
inv2 = sm.OLS(Y, X).fit()
inv2.fittedvalues # same approx solution as raw computation
```

```
array([0.99860208, 0.99970045, 1.00169257])
```

4.1.4 - R OLS

Using the same technique as done in Python, the pseudo inverse is used to find the ordinary least squares solution. This is done with the R function `pinv()` from the pracma package. More information on the `pinv()` function can be found on the pracma documentation page [10].

```
# set up with same "random" data as Python example
X <- matrix(c(0.67459005, 0.48042258, 0.64218228, 0.33554957, 0.86050783, 0.429431), ncol = 2)
Y <- rep(1, 3)
print(X)
print(Y)
```

```
          [,1]      [,2]
[1,] 0.6745901 0.3355496
[2,] 0.4804226 0.8605078
[3,] 0.6421823 0.4294310
[1] 1 1 1
```

```
inv <- pinv(X)
S <- inv %*% Y
print(X %*% S) # approximate solution to y
```

```
          [,1]
[1,] 0.9986021
[2,] 0.9997004
[3,] 1.0016926
```

These results are consistent with the computations done in Python.

4.2 - Generalized Least Squares (GLS)

GLS makes use of the psd inverse calculations and matrix multiplication functions. For solution S, data matrix X, solution space Y and PSD matrix Σ:

$$
\begin{aligned}
XS &= Y \\
X^T \Sigma^{-1} XS &= X^T \Sigma^{-1} Y \\
S &= (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} Y
\end{aligned}
$$

Since Σ is PSD, then Σ^{-1} is PSD and hence $(X^T \Sigma^{-1} X)$ is also PSD. Thus using cholesky decomposition for $\Sigma = L_1^T L_1$ and the inner matrix $(X^T \Sigma^{-1} X) = X^T (L_1^{-1} L_1^{-T}) X = L_2^T L_2$ the GLS computation then becomes:

$$
\begin{aligned}
S &= (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} Y \\
&= (L_2^{-1} L_2^{-T}) X^T (L_1^{-1} L_1^{-T}) Y
\end{aligned}
$$

4.2.1 - Python GLS

The shortcuts discussed above for psd matrices can be used directly in the GLS computation. One can implement a version using cholesky decomposition methods, as well as a "pure" version which does not and finally a built-in method using the statsmodels module.

```
# A fast generalized least squares solution for data matrix X, solution space Y and PSD matrix sigma.
# The function computes X(X^Tsigma^{-1}X)^{-1}X^Tsigma^{-1}Y
# Notes: -> sigma must be PSD
#        -> dimensions of X, Y and sigma must agree for matrix multiplication
def chole_gls(X, Y, sigma):

    # Find first inverse sigma^{-1}
    L = la.solve_triangular(np.linalg.cholesky(sigma), np.eye(len(sigma)), lower=True)

    # Find next inverse (...)^{-1}
    inner = X.T @ np.matmul(L.T,L)
    L2 = la.solve_triangular(np.linalg.cholesky(inner @ X), np.eye(len(X)), lower=True)

    # Return gls
    return np.matmul(L2.T,L2) @ inner @ Y


# A pure generalized least squares solution for data matrix X, solution space Y and PSD matrix sigma.
# The function computes X(X^Tsigma^{-1}X)^{-1}X^Tsigma^{-1}Y
# Notes: -> sigma must be PSD
#        -> dimensions of X, Y and sigma must agree for matrix multiplication
def pure_gls(X, Y, sigma):

    # Find first inverse sigma^{-1}
    s_inv = la.inv(sigma)

    # Return gls
    return la.inv(X.T @ s_inv @ X) @ X.T @ s_inv @ Y
```

A small GLS example is performed below using a matrix of size $\mathbb{R}^{3\times3}$ where all three methods are tested.

```
# Data
x = np.random.uniform(0,1, size = (3,3))
y = np.array([1.,1,1])
# PSD sigma matrix
s = np.random.uniform(0,1, size = (3,3)) # symmetric psd
s = s + s.T - np.diag(s.diagonal())
s = np.matmul(s,s.T)/3
x,y,s
```

```
(array([[0.05154553, 0.33478251, 0.68424231],
        [0.7103685 , 0.42704954, 0.5659001 ],
        [0.6738195 , 0.29515479, 0.28922683]]),
 array([1., 1., 1.]),
 array([[1.14032497, 1.0809451 , 1.22701868],
        [1.0809451 , 1.33186762, 1.04385891],
        [1.22701868, 1.04385891, 1.86092279]]))
```

```
soln = pure_gls(x, y, s)
soln_chole = chole_gls(x,y,s)
soln_sm = sm.GLS(y, x, s).fit()
x @ soln, x @ soln_chole, soln_sm.fittedvalues # 3 solutions all equal to y
```

```
(array([1., 1., 1.]), array([1., 1., 1.]), array([1., 1., 1.]))
```

One can observe that the results are all equivalent for each method. To view the difference in computational run times for each of these methods, a small experiment is run using varying sized matrices.

```
N = np.array([2, 10, 50, 500, 1000, 5000, 10000])
n = len(N)

# data
X = np.random.uniform(0,1, size = (N[n-1], N[n-1]))
Y = np.ones(N[n-1])
# psd sigma
sigma = np.random.uniform(0,1, size = (N[n-1], N[n-1]))
sigma = 0.5*(sigma + sigma.T)
sigma = sigma + N[n-1] * np.eye(N[n-1])

time_normal = []
time_chole = []
time_sm = []

for i in range(n):

    t1 = time.time()
    sol1 = pure_gls(X[0:N[i], 0:N[i]], Y[0:N[i]], sigma[0:N[i], 0:N[i]])
    time_normal.append(time.time()-t1)

    t2 = time.time()
    sol2 = chole_gls(X[0:N[i], 0:N[i]], Y[0:N[i]], sigma[0:N[i], 0:N[i]])
    time_chole.append(time.time()-t2)

    t3 = time.time()
    sol3 = sm.GLS(Y[0:N[i]], X[0:N[i], 0:N[i]], sigma[0:N[i], 0:N[i]]).fit()
    time_sm.append(time.time()-t3)
```

Computational Runtime (seconds)

N for NxN sized matrix	Pure GLS	Cholesky GLS	statsmodels GLS
10	0.000081	0.000589	0.000398
50	0.000610	0.001257	0.001597
500	0.630047	0.020646	0.497284
1000	3.008091	0.069500	2.308699
5000	11.976236	3.500787	65.164299
10000	21.954963	15.347694	545.324912

Note: these runtimes will vary depending on the machine one uses but we can expect the overall trends to stay the same. One can observe how much slower the built-in statsmodels function `sm.GLS()` is compared to the other two methods. This can be attributed to the statsmodels function performing additional tasks within the `GLS()` call. These tasks include calculating residuals, fitted values and other descriptive summary results. If your goal is to produce nice results and are not concerned with run time then using this function may be desired. If run time is a concern then building GLS functions from scratch is clearly advantageous. These raw functions can be improved by implementing a cholesky decomposition method, which was shown above to compute solutions the fastest.

4.2.2 - R GLS

The same process for GLS computation can be carried out in R, making use of the same psd matrix tricks. It should be noted that matrix multiplication is slightly slower in R compared to NumPy.

```
# A fast generalized least squares solution for data matrix X, solution space Y
# and PSD matrix sigma. The function computes (X^Tsigma^{-1}X)^{-1}X^Tsigma^{-1}Y
# Notes: -> sigma must be PSD
#        -> dimensions of X, Y and sigma must agree for matrix multiplication
chole_gls <- function(X,Y,sigma) {

  # sigma inverse
  Linv <- backsolve(chol(sigma), diag(ncol(X)))

  # inner inverse
  inner <- crossprod(tcrossprod(Linv), X)
  L2 <- backsolve(chol(crossprod(inner, X)), diag(ncol(X)))

  # return gls soln
  tcrossprod(tcrossprod(L2), inner) %*% Y
}


# A pure generalized least squares solution for data matrix X, solution space Y
# and PSD matrix sigma. The function computes (X^Tsigma^{-1}X)^{-1}X^Tsigma^{-1}Y
# Notes: -> sigma must be PSD
#        -> dimensions of X, Y and sigma must agree for matrix multiplication
pure_gls <- function(X, Y, sigma) {

  # sigma inverse
  inv1 <- solve(sigma)

  # inner inverse
  inv2 <- solve(t(X) %*% inv1 %*% X)

  # return gls
  inv2 %*% t(X) %*% inv1 %*% Y
}
```

Similar to what was done above for Python, a small GLS example is performed below using a matrix of size $\mathbb{R}^{3\times3}$. Here the cholesky method is implemented and compared to a "pure" implementation that uses no decomposition.

```
x <- matrix(runif(9), ncol = 3)
y <- c(1,1,1)
s <- matrix(runif(9), ncol = 3)
s <- 0.5*(s + t(s))
s <- s + 3*eye(3)
print(x)
print(y)
print(s)
```

```
          [,1]       [,2]      [,3]
[1,] 0.9049013 0.14480767 0.5152959
[2,] 0.7297637 0.40634528 0.2310326
[3,] 0.2044630 0.04217148 0.5226685
[1] 1 1 1
          [,1]      [,2]      [,3]
[1,] 3.4516124 0.5254392 0.7363178
[2,] 0.5254392 3.2415333 0.8283556
[3,] 0.7363178 0.8283556 3.3701677
```

```
soln1 <- pure_gls(x,y,s)
soln2 <- chole_gls(x,y,s)
print(x %*% soln1)
print(x %*% soln2) # same solution equal to y
```

```
     [,1]
[1,]    1
[2,]    1
[3,]    1
     [,1]
[1,]    1
[2,]    1
[3,]    1
```

One can observe the equivalence os the solutions for each method. To view the difference in computational run times for each of these methods, a small experiment is run using varying sized matrices.

```
scales <- c(2, 10, 50, 500, 1000, 5000, 10000)
n <- length(scales)
time_solve <- numeric(n)
time_chol <- numeric(n)
X <- matrix(runif(scales[n]^2, 0, 1), nrow = scales[n], ncol = scales[n])
Y <- rep(1, scales[n])

s <- matrix(runif(scales[n]^2, 0, 1), nrow = scales[n], ncol = scales[n])
s <- 0.5 * (t(s) + s)
s <- s + scales[n] * eye(scales[n])

for (i in 1:n) {
  N <- scales[i]

  t1 <- Sys.time()
  soln1 <- pure_gls(X[1:N, 1:N], Y[1:N], s[1:N, 1:N])
  time_solve[i] <- Sys.time() - t1

  t3 <- Sys.time()
  soln2 <- chole_gls(X[1:N, 1:N], Y[1:N], s[1:N, 1:N])
  time_chol[i] <- Sys.time() - t3
}
```

A matrix: 6 x 3 of type dbl

N for NxN sized matrix	Pure Implementation	Cholesky Implementation
10	0.000717	0.000070
50	0.001714	0.000257
500	1.063931	0.069808
1000	2.827480	0.168958
5000	13.069145	3.751185
10000	33.688865	16.694152

Note: these runtimes will vary depending on the machine one uses but we can expect the overall trends to stay the same. The results are similar to what was found in Python. Taking advantage of a cholesky decomposition when computing a GLS solution in R clearly leads to faster run times. The difference in run time looks to get larger as the size of the system increases.

[9] Virtanen, P. et al., 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, [link] (https://scipy.org/)
[10] Hans W. Borchers, 2022, Package 'pracma' (Practical Numerical Math Functions), [link] (https://CRAN.R-project.org/package=pracma)
[11] Seabold, Skipper, and Josef Perktold, 2010. statsmodels: Econometric and Statistical Modeling with Python, [link] (https://www.statsmodels.org)