Some Notes on Matrix Inverse Functions and Fitting a Line to Data

Ellianna Abrahams, UC Berkeley

March 2021

The following are some notes that might be useful in understanding the role of matrix inversion in fitting a line to data, more formally known as linear regression. I hope that they are useful, but I do recognize that they can be lengthy. The TL;DR of this set of notes is Section 3, which reviews the best practices for fitting a line to functions of the form $Y = X\beta$, *i.e.* the methodology for applying the matrices you wrote out in Question 5 from Lab 1 to answering Question 6 there. This methodology will again be useful in answering Lab 2.

1 Does a Rectangular Matrix Have an Inverse?

In your linear algebra classes you may have learned that squareness is one of the necessary conditions for the inverse of a matrix to exist. This is true, but it still doesn't prevent us from taking the inverse of rectangular matrices, if we take some clever steps. In this section, we'll work through some of the equations to take the "inverse" of a rectangular matrix under the right conditions.

Before we dive into how this is achieved for a rectangular matrix, let's generally define the inverse of a matrix. The inverse of a matrix A is a matrix that, when multiplied by A, results in the identity matrix. The notation for this inverse matrix is $A^{˘1}$, *i.e.* $(A^{-1})A = I$. While the strict conditions for this inverse to exist are that A is square (*i.e.* $m = n$), and that the determinant of A is nonzero, if stick to our general definition of A^{-1}, an inverse exists if there is a matrix that when multiplied against A, results in I. By this more loose definition, we can then find the "inverse" of A even when $m \neq n$, under the right conditions.

More specifically, when there are more rows than columns in A, $m > n$, we can find the one-sided left-inverse matrix. Alone A is rectangular, and therefore not invertible, but we can manipulate A to allow for its inverse function to exist. We start by multiplying A by its transpose, which will result in a square matrix. As long as the resultant matrix is full rank[1], its inverse will exist. Let's write this out.

$$\underbrace{(A^T A)}_{n\times n}{}^{-1}\big(\underbrace{A^T}_{n\times m}\overbrace{A}^{m\times n}\big) = \underbrace{I}_{n\times n}$$

While we are unable to break up the expression in the first parentheses, as it only exists *because* A is multiplied by its transpose, we can break up the expression in the second parentheses, and this will give us our "inverse" for the rectangular matrix A.

$$\underbrace{\big((A^T A)^{-1}A^T\big)}_{\text{Left Inverse}}A = I \tag{1}$$

[1] A matrix is full rank *iff* its determinant exists and is nonzero.

This grouping of $(A^TA)^{-1}A^T$ conveniently returns the identity matrix when multiplied with A, making it an inverse by definition. It is formally called the left inverse, because it must be applied to the left side of A in order to obtain the correct dimensions. Since we need A^TA to be full rank for its inverse to exist, this means that a left-inverse exists for A *iff* A is full column rank.

A similar expression can be written when A has more columns than rows ($m < n$). It is left to the reader as an exercise to test why this row and column configuration requires a right inverse instead, given by the following expression.

$$A\underbrace{\left(A^T(AA^T)^{-1}\right)}_{\text{Right Inverse}} = I \tag{2}$$

Since AA^T must be full rank for an inverse to exist, we require in this case that A is full row rank for a right inverse to exist.

This means that we can use the left inverse to solve equations of the form $AX = Y$ via the following sequence.

$$\begin{aligned} AX &= Y \\ (A^TA)^{-1}A^TAX &= (A^TA)^{-1}A^TY \\ IX &= (A^TA)^{-1}A^TY \quad (\textit{using Eqtn. 1}) \\ X &= (A^TA)^{-1}A^TY \end{aligned}$$

2 Applying the Left Inverse to Fit a Line to Data

The left inverse ends up being a handy tool for fitting a line to data, since fitting a line requires us to solve equations of the form $X\beta = Y$, where X is the independent variable, Y is the response variable and β are the parameters describing the line. Using the math we stepped through in Section 1, we can estimate $\hat{\beta}$ using the left inverse, and in this section we'll show how.

As an example, we'll set up a single variable line equation using regression notation, with β_1 symbolizing the slope of the line and β_0 symbolizing the y-intercept. For people more familiar with standard line notation, $\beta_0 = b$ and $\beta_1 = m$.

$$y = \beta_0 + \beta_1 x$$

Assuming that both y and x are vectors of length n, we can rewrite this equation in matrix form.

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} \beta_0 + \beta_1 X_1 \\ \beta_0 + \beta_1 X_2 \\ \vdots \\ \beta_0 + \beta_1 X_n \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}}_{Y_{n\times 1}} = \underbrace{\begin{bmatrix} 1 & X_1 \\ 1 & X_2 \\ \vdots & \\ 1 & X_n \end{bmatrix}}_{X_{n\times 2}} \cdot \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}}_{\beta_{2\times 1}}$$

In this way, we have rewritten the equation of a line as a solvable inverse problem. When we collect data we are often interested in the underlying functions that describe the relationships between variables. We collect measurements of X and Y on a finite sample. Assuming that our finite sample is representative, we'd like to use our measurements to estimate the parameters, β, that describe the underlying population. This estimation of our parameter matrix is symbolized by $\hat{\beta}$, and using the left-inverse from Section 1, we can write it as the following.

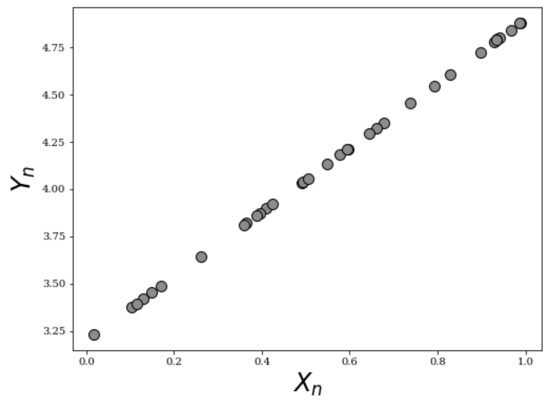
$$\begin{aligned} X\hat{\beta} &= Y \\ (X^TX)^{-1}X^TX\hat{\beta} &= (X^TX)^{-1}X^TY \\ \hat{\beta} &= (X^TX)^{-1}X^TY \end{aligned} \tag{3}$$

This means that if our data has no scatter (intrinsic or systematic), we can directly estimate $\hat{\beta}$, using Eqtn. 3. We can simulate some data in `python` to show how this works.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

n = 35
beta0_truth = 3.2
beta1_truth = 1.7

x0 = np.ones(n)
x1 = np.random.uniform(0.0, 1.0, n)
y = beta0_truth + beta1_truth*x1

X = np.column_stack((x0, x1))
Y = y.reshape(n,1)

print(X.shape)
print(Y.shape)

OUTPUT: (35, 2)
        (35, 1)
```

If we plot this data, we can see that the data falls exactly on the line that we defined, with uniform randomness in its placement on the x-axis.

```
# Use Eqtn. 3 to estimate $\hat{\beta}$
# $\hat{\beta} = (X^TX)^{-1}X^TY$

beta_hat = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), Y)

beta0_hat = beta_hat[0]
beta1_hat = beta_hat[1]

print("beta0_hat = {}".format(beta0_hat) + "\n" \
      + "beta1_hat = {}".format(beta1_hat))

OUTPUT: beta0_hat = [3.2]
        beta1_hat = [1.7]
```



Because we simulated data that fell perfectly on the line without any intrinsic or systematic scatter, we were able to estimate $\hat{\beta}$ directly. The only reason we are careful to still call this measure an estimator is because we have taken initial x and y measurements from a finite sample.

In reality the data we take always has some scatter to it. Even the most perfectly collected data is limited to quantum fluctuations in photon arrival times and within the electrons we use to detect the signals in our instruments. This means that we need some way to measure the underlying trend within a noisy sample. So how do we estimate $\hat{\beta}$ when dealing with real data?

3 The Method of Least Squares

The Method of Least Squares is an elegant way to optimize finding a useful estimation of $\hat{\beta}$ from the data that we have in real life: data with scatter[2]. The most important aspect of Least Squares for our purposes is the calculation of the parameters in $\hat{\beta}$. We'll step through the calculation for those parameters assuming again that our $\hat{\beta}$ matrix is 2×1, *i.e.* $\beta_0 = b$ and $\beta_1 = m$ in standard notation.

We optimize our calculation of each $\hat{\beta}$ by minimizing the least squares error, e, between our estimate of the response variable, $\hat{Y} = X_{sampled}\hat{\beta}$, and our measurements of the response variable, Y. This is hard to visualize in words, so let's write it out, where i is the index of our sampled values.

$$\begin{aligned} e &= \sum_{i=1}^{n} \big(Y_i - (X_i\hat{\beta}) \big)^2 \\ &= \sum_{i=1}^{n} \big(Y_i - \hat{Y}_i \big)^2 \end{aligned}$$

In our case, with a 2 parameter $\hat{\beta}$, this breaks down into two equations – the derivatives *w.r.t.* $\hat{\beta}_0$ and $\hat{\beta}_1$ – that we set equal to zero and solve.

$$\frac{\partial e}{\partial \hat{\beta}_0} = \frac{\partial e}{\partial \hat{\beta}_1} = 0 \tag{4}$$

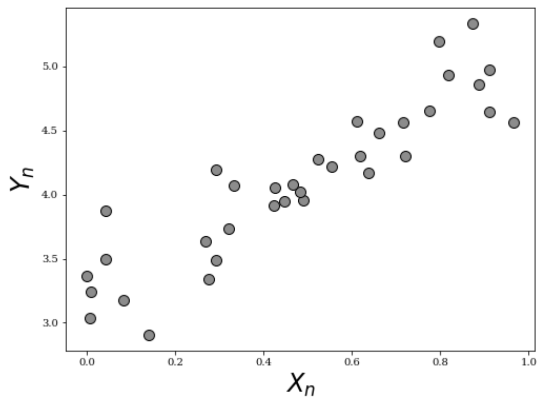
When we have just two parameters this can be solved by hand, and is left as an exercise for the reader. It

[2]At this point the full derivations for Least Squares are beyond this discussion, in order to get to our intended goal of applying a linear fit to our data. However the derivations for least squares regression are elegant and great for developing fluency in statistical thinking. If you are interested in stepping through them, resources will be included at the end of this document.

is easy to imagine though that this quickly gets complicated when more than one independent variable, or covariate, contributes to our calculation of $\hat{Y}$. Lucky for us, `python` comes to the rescue with the set of `linalg` functions[3] in `numpy`, and we'll solve a simulated example using those functions here, this time adding normally distributed scatter to the response variable.

```
# Define sample size and true parameters
n = 35
beta0_truth = 3.2
beta1_truth = 1.7

x0 = np.ones(n)
x1 = np.random.uniform(0.0, 1.0, n)
y = beta0_truth + beta1_truth*x1

# Add normally distributed scatter
e = np.random.normal(0.0, 0.25, n)
y += e

X = np.column_stack((x0, x1))
```

Plotting this data, we can see that y is dependent on x, but the scatter makes the underlying parameters describing the dependency difficulty to guess by just calculating the slope between two points.



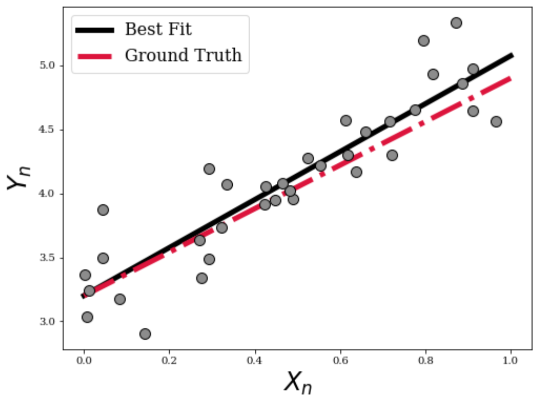
However, because we can see that y is dependent on x in what appears to be a linear manner, this makes our data a great candidate for running least squares regression. Here's how we do that in `python`.

```
# Use np.linalg.lstsq to solve for β̂
# Note from the documentation that the shape of y
# needed for the function is different
# than the shape we used for y
# in the previous section
beta_hat = np.linalg.lstsq(X, y, rcond=None)[0]

beta0_hat = beta_hat[0]
beta1_hat = beta_hat[1]
```

[3] https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html

```
print("beta0_hat = {}".format(beta0_hat) + "\n" + "beta1_hat = {}".format(beta1_hat))

OUTPUT: beta0_hat = 3.2010521046175424
        beta1_hat = 1.873499949507801
```

Adding our fit to the plot, we can see from our printed outputs that our Least Squares estimation did not get $\hat{\beta}$ exactly, like it did in the previous section when our data had no scatter. Considering the strength of the scatter added to the data and our small sample size, our $\hat{\beta}$ is pretty close!



What do you think needs to be done to make the fit even better? Check out the companion notebook for a couple of directions.

In these notes we used simulated data, but the real power of Least Squares fitting lies in its usefulness for working with real-world data. In Lab 1, Question 5 suggested that if we know the angular frequency for a given fourier sum, we can recast the fourier fit as a problem of the form $Y = X\beta$, i.e. the exact form that we can solve with a Least Squares fit. If you used another fitting method to solve Question 6, give least squares a try! See if it is more efficient than the method you used by timing the fit, and in particular, check how the cross-validation performs on the Least Squares method compared to your initial method. Did Lease Squares find a higher or lower fourier order, K? Which one do you think is more accurate and why?

Resources

Left and Right Matrix Inversion:
https://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/positive-definite-matrices-and-applications/left-and-right-inverses-pseudoinverse/MIT18_06SCF11_Ses3.8sum.pdf

The Method of Least Squares Derived:
https://www.amherst.edu/system/files/media/1287/SLR_Leastsquares.pdf

Documentation for the `numpy.linalg` Package:
https://numpy.org/doc/stable/reference/routines.linalg.html