# **Least squares.**

It can happen that the matrix equation ***Ax*** *=* ***b*** has no exact solution. The most common reason is that the system of equations is *overdetermined*: this means that there are more equations than unknowns, and therefore ***A*** has more *rows* than *columns* (*m* > *n*). For example:

***A x b***

The vector ***b*** has 7 elements and thus belongs to the 7-dimensional space **R7**. A complete representation of this space requires a *basis* of 7 vectors, but we know that, if there was an exact solution, ***b*** would alsobelong to the *column space* of ***A***, which is a 2-dimensional subspace of **R7**. Since that space is only a *plane* in a 7-dimensional space, it covers a minuscule fraction of **R7**, and thus chances that any vector ***b*** of **R7** will be on that plane are very slim. In fact, for the matrix equation shown above, there is no combination of the columns of ***A*** that would produce ***b***.

If there was an exact solution to ***Ax*** *=* ***b*** it would mean that ***b - Ax*** *=* ***0***. On the other hand, just as we know there is no solution, we can easily recognize that there exists an infinite number of values of ***x*** such that ***b - Ax*** *=* ***e*** with ***e*** being a vector different from **0**. Thus, the *best* we can do is to search for an *optimal* vector ***x*** (often denoted as **)** such that ***e*** is as close as possible to **0**. To find the 'smallest' value of ***e*** means to *minimize* the value of its *length*, ||***e***|| or, more simply, its *squared length* ||***b - Ax***||2 = ||***e***||2 = ***eTe***: in other words we are looking for a



***least squares solution***

for the matrix equation ***Ax*** *=* ***b*** in the form of an

***exact solution***

for the matrix equation , with ***p*** being a vector as similar as possible to ***b***.

We have already encountered this situation in our discussion of *projections*, and we know already that, if there is no solution for:

provided the columns of are linearly independent (that is, is invertible), we can still find a solution for the system:

which is the same as finding a solution for the system , in which ***p*** is the projection of ***b*** onto the *column space* of ***A*** (that is, the vector of the ***A*** space that is most similar to ***b***).

We can look at the process of finding a *least squares* solution from the point of view of the *fundamental theorem of linear algebra* (CHAPTER 4). When we talked about the 4 spaces of matrix ***A*** (*m* x *n*) with full column rank (all *n* columns linearly independent) we described how a vector ***x*** from the **Rn** space is brought into **Rm** by splitting it into its 2 projections into the *row* and *null* space of ***A***. Here we have a similar situation, but we are splitting ***b*** instead of ***x***.



Notice how **N*(A)*** is very small (just one point at the origin), because all the columns of ***A*** are linearly independent. Therefore ***ATA*** is*n x n* **,** full rankand invertible. The equation:

is the ***normal equation***, and (often represented in the literature with the symbol ) provides the ***B****est* ***L****inear* ***U****nbiased* ***E****stimate* (BLUE) of in.

Remember that the error, ***e*** *=* ***b*** *-* ***p***, is unavoidable. Since ***e*** is in the *left null space of* ***A***(*null space* of ***AT***, **ATe** = **0**), it is perpendicular to the *column space*of ***A*,** and thus it isthe shortest vector (smallest value of ||***e***||2 = ***eTe***) from the subspace **C(*A*)**tothe point ***b***. Therefore, ***p*** is the *'most similar'* vector to ***b*** that allows a solution.

The most common application of *least squares* is the fitting of a line to a set of points. It helps understanding how this fitting is done to consider first a set of points perfectly aligned. For example, we start by writing the equations (of general form ***y*** = b + a***x***) representing each point:

y1 = int + slope \* x1

y2 = int + slope \* x2

y3 = int + slope \* x3

y4 = int + slope \* x4

y5 = int + slope \* x5

y6 = int + slope \* x6

y7 = int + slope \* x7

We are given the values of all the x's and y's (these are the coordinates of each point). The unknowns (***u***)in these equation are the intercept (int) and the slope. Therefore, we can re-write the equations as:

1\*int + x1\*slope = y1

1\*int + x2\*slope = y2

1\*int + x3\*slope = y3

1\*int + x4\*slope = y4

1\*int + x5\*slope = y5

1\*int + x6\*slope = y6

1\*int + x7\*slope = y7

and representing in matrix notation:

***A u b***

To create a simple example of *least squares* we start with the coordinates of 7 perfectly aligned points by assigning an intercept and a slope, and then we add some random noise (proportional to y) to the y coordinates (the ***b*** vector) to simulate an experimental error with a normal distribution:

⇒

***A u b***

xvec = [1:7]';

int = 5;

slope = 10;

yvec = int + slope\*xvec;

lsq\_plot = figure

We get the handles for each variable plotted so we can then select specific handles to use in the legend:

h(1) = plot(xvec,yvec,'+r');

hold on

set(gca,'Xlim',[0 8],'Ylim',[0 100]);

Here we create a large array (1,000 samples) of possible results of an experiment so we can calculate its covariance matrix (CHAPTER 10), ***V***. We choose between two options for the standard deviation: either *constant* (set to the mean of some fraction of the *yvec* values) [this is the so-called *homoskedasticity condition*] or proportional to the *yvec* values [*heteroskedasticity condition*]:

sigma\_vec = {'constant sigma' 'proportional sigma'};

sigma = sigma\_vec(2);

nsamples = 1000;

sigma\_y = 0.2\*yvec;

mean\_sigma\_y = mean(sigma\_y);

if strcmp(sigma, 'constant sigma')

noise = normrnd(zeros(m,nsamples),mean\_sigma\_y(ones(m,1),ones(nsamples,1)));

elseif strcmp(sigma,'proportional sigma')

noise = normrnd(zeros(m,nsamples),sigma\_y(:,ones(nsamples,1)));

end

yvec\_m = yvec(:,ones(nsamples,1)) + noise;

V\_noise = cov(noise');

V = cov(yvec\_m');

Then only one of these samples is further analyzed.

yvec\_ind = randi(nsamples);

yvec\_n = yvec\_m(:,yvec\_ind);

h(2) = plot(xvec,yvec\_n,'sb');

legend('perfect data','data with noise','Location','best')

xlabel('x'),ylabel('y')

Now we can plot the perfectly aligned points and the points with random noise. Clearly, the ***b*** vector corresponding to the data with noise is not in the *column space* of ***A***, but we can easily find the *projection* ***p*** of ***b*** onto **C*(A)***. We can take the columns of ***A*** as the *basis* for **C*(A)***. First, we calculate the *projection matrix* ***P***:

A = [ones(7,1) xvec]

[m,n] = size(A)

P = A\*inv(A'\*A)\*A'

P = A/(A'\*A)\*A'

P', P\*P

which is a *symmetric 7x7* matrix. Knowing ***P*** we can calculate the projection ***p*** of ***b*** onto ***C(A)***:

p = P\*yvec\_n

figure(lsq\_plot)

h(3) = plot(xvec,p,'og');

and now we are ready for an *exact solution*:

However the same result can be obtained by directly finding the exact solution of the matrix equation:

Recalling that is symmetric and invertible we derive :

u = (A'\*A)\A'\*yvec\_n

which is the same as finding the value of for which we achieve the minimum of:

under the *constraint* of orthogonality between the column space of and . In fact we can rearrange the normal equation as:

*Regression* is the process of fitting models to data. *Linear regression* assumes that the relationship between the independent variable *xi* (the *predictor* or *regressor*) and the dependent variable *yi* (the *response*) is linear, i.e.:

*yi* = *int* + *slope* *xi*.

The type of the *linear regression* afforded by the application of the *normal equation* is usually referred to as the *ordinary least-squares* (**OLS**) *estimates* of the *regression coefficients*  (often referred to in the literature as ), in this case the intercept and slope parameters, which in MATLAB can also be derived simply using the backslash operator \:

u = A\yvec\_n

We can now plot the *least squares* solution:

yvec\_ls = u(1) + u(2)\*xvec;

figure(lsq\_plot)

h(4) = plot(xvec,yvec\_ls,'-m');

legend('perfect data', 'data with errors','projected data','least squares solution')

Notice how the straight line derived from this solution is *close*, but not quite identical to the real Y value (red crosses). What is the error ***e*** in our solution? We know that it is the projection of ***b*** onto the *left null* space of ***A*** (the *null space* of ***AT***)! We can plot also this error together with our solution.

N\_at = null(A');

P\_ln = N\_at \* inv(N\_at' \* N\_at) \* N\_at';

P\_ln = N\_at / (N\_at' \* N\_at) \* N\_at';

The same matrix can be obtained by simply subtracting ***P*** from the identity matrix ***I***.

I = eye(m)

P\_ln = I-P;

Now we can add the error bars to our plot.

error = P\_ln\*yvec\_n;

figure(lsq\_plot)

for i = 1:nobs

x = [xvec(i),xvec(i)];

y = [yvec\_n(i),yvec\_n(i)-error(i)];

h(4+i) = line(x,y,'LineWidth',3,'Color','g');

end

h(12) = plot(xvec,yvec,'--r');

legend({'perfect data','data with noise','projected data', ...

'least squares solution','errors'},'Location','best')

The important thing to remember is that the *sum of squared errors* (*sse*) , commonly reported by programs that carry out least square analyses, is only a measure of how well our model (the purple line) fits the experimental data (the blue squares), which has noise, and NOT a measure of how close our MODEL is to what REAL DATA would look like if the experimental noise could be removed (represented by the dashed red line).

sse = error'\*error

sse = yvec\_n'\*P\_ln\*yvec\_n

Many programs will report the *standard deviation* ***σ*** or the mean squared error, *mse* or :

where *m*is the number of experimental points and *n*is the number of *parameters* (the unknowns of the normal equation).

sigma = sqrt(sse/(m-n))

mse = sse/(m-n)

Another criterion to evaluate the quality of the fit is the *Rsquare* value calculated as:

(*Fraction of Unexplained Variance*)

where is the sum of squares about the mean and is often called the *total sum of squares* (*sst*):

c\_yvec = yvec\_n - mean(yvec\_n);

sst = c\_yvec'\*c\_yvec;

rsquare = 1-sse/sst

where ***b*** is the vector of experimental observations and is its mean value (therefore represents what is called a *centered* ***b*** vector). The closer the *Rsquare* value is to 1 the more reliable is the fit. The ratio:

is the fraction of *unexplained variance* (FUV) since it is the ratio between the unexplained variance (due to errors) and the total variance of the data. Consequently, is the fraction of *explained variance*:

where is often called the *sum of squares of the regression* (*ssr*):

c\_p = p - mean(yvec\_n);

ssr = c\_p’\*c\_p;

rsquare = ssr/sst

In the **OLS** case the *Rsquare* value is simply the square of the *correlation coefficient* between the observed data, , and the model data, :

rsquare = (corr(yvec\_n,p))^2

rsquare = ((c\_yvec/norm(c\_yvec))'\*((p-mean(p))/norm(p-mean(p))))^2

A modification of the *Rsquare* to account for the number of variables involved in the fit, leads to the *adjusted Rsquare*, sometimes represented with the symbol (pronounced *Rbar squared*):

adj\_rsquare = 1-sse\*(m-1)/(sst\*(m-n))

*Rsquare* can take on any value between 0 and 1, with a value closer to 1 indicating that a greater proportion of variance is accounted for by the model. For example, an *Rsquare* value of 0.8234 means that the fit explains 82.34% of the total variation in the data about the average. The *adjusted Rsquare* statistic is generally the best indicator of the fit quality when we compare two models that are *nested* — that is, a series of models each of which adds additional coefficients to the previous model. The *adjusted Rsquare* statistic can take on any value less than or equal to 1, with a value closer to 1 indicating a better fit. Negative values can occur when the model contains terms that do not help to predict the response.

Regression *t* and *F* statistics.

While in the example above we have only 1 predictor (the *xvec* variable) for the response (the *yvec* variable), nothing prohibits us from having a multiple linear regression (MLR) model that relates a variable (response) to *p* variables (multiple predictors). MLR has the form:

where we assume that the have a normal distribution with mean 0 and constant variance. In matrix notation an MLR model is represented as:

response design matrix slope error

vector vector vector

The regression coefficients (the slope vector) are determined as usual by employing the normal equation:

and the residual is:

One of the main goals of fitting a regression model with multiple predictors is to determine which predictor variables are truly related to the response. This can be formulated as a set of hypothesis tests. For each predictor variable , we test the *null hypothesis* against the *alternative hypothesis* . We recall here that if is an estimator of parameter in some statistical model, then a *t-*statistic for this parameter is a quantity of the form:

where is the *estimated standard deviation* ***σ*** of the distribution of each coefficient (see later on in this chapter):

By default, statistical packages report *t-*statistic with in order to test the significance of a regressor:

For given degrees of freedom, , where is the number of observations, and is the number of regression coefficients (including the intercept), the probability distribution of *-*values, when the *null hypothesis* is true,is the Student’s *t*-distribution. The *p*-value is the cumulative probability of finding a *t-*value equal or larger than that observed. This indirectly reflects the probability of the null hypothesis to be true.

Thus, to obtain the *p*-value for each regressor, first we standardize the slope estimates:

se = sqrt(diag(mse\*inv(A'\*A)))

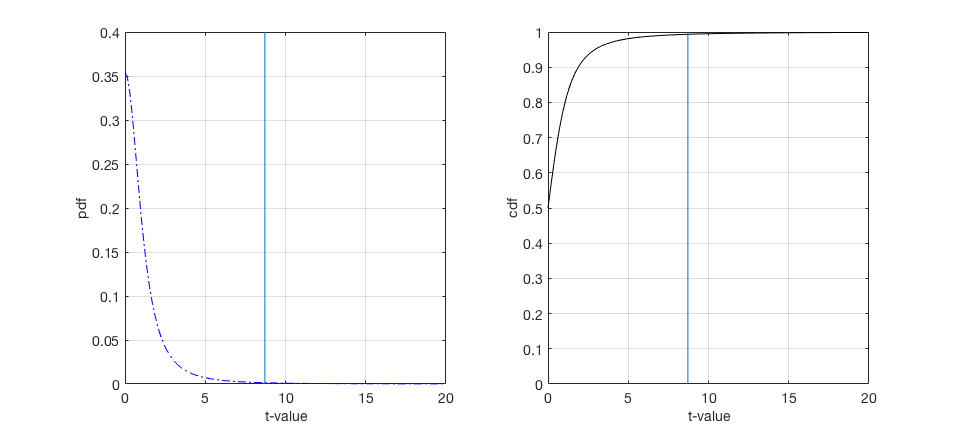
t = u(2)/se(2)

t = 8.7167

We can visualize the probability density function (*pdf*) and the cumulative distribution function (*cdf*) for the *t*-statistics:

nu = m-n

x = [0:.1:20];y1 = tpdf(x,n); y2 = tcdf(x,n);

T\_statistics = figure;

set(gcf,'Unit','Normalized',…

'Position',[0.2 0.5 0.6 0.4])

subplot(1,2,1);

plot(x,y1,'Color','blu','LineStyle','-.');

xlabel('t-value'); ylabel('pdf')

ca = gca;line([t t],ca.YLim);

subplot(1,2,2);

plot(x,y2,'Color','black','LineStyle','-');

xlabel('t-value'); ylabel('cdf')

line([t t],[0 1]);

Then we find the complement to 1 of the value of the *cdf* for each *t* value based on the known degrees of freedom . For the simple case of OLS discussed above, we have only 1 *t value* (a vertical lines marks this value in the *pdf* and *cdf*).

p\_t = 1-tcdf(t,nu)

p\_t = tcdf(t,nu,'upper')

p\_t = 1.6449e-04

We can also test for the *null hypothesis* ‘‘ (*reduced model* having only the intercept) against the *alternative hypothesis* ‘’ (*full model*). In this case, instead of *t-statistics,* we use *F-statistics.* An *F-test* involves a comparison between the sum of square error of the reduced model having only the intercept term (this is the same as *SST* or *total sum of square*) and that of the full model (*SSE*).

If *SSE* is close to *SST*, then the variation around the estimated full model regression function is almost as large as the variation around the estimated reduced model regression function. In that case, it makes sense to use the simpler *reduced model*.

Conversely, if *SSE* and *SST* differ greatly, then the additional parameter(s) in the full model substantially reduce the variation around the estimated regression function. In this case, it makes sense to go with the larger *full model*.

The general linear *F-statistic* is a function of the difference in the error between the two models:

Since (*regression sum of squares*) we derive the *F-*statistics as:

which is clearly distinct from the *fraction of explained variance* :

If the *null hypothesis* is true has an *F distribution*, , with numerator degree of freedom and denominator degrees of freedom. This distribution is obtained as the ratio of two distribution with and degrees of freedom:

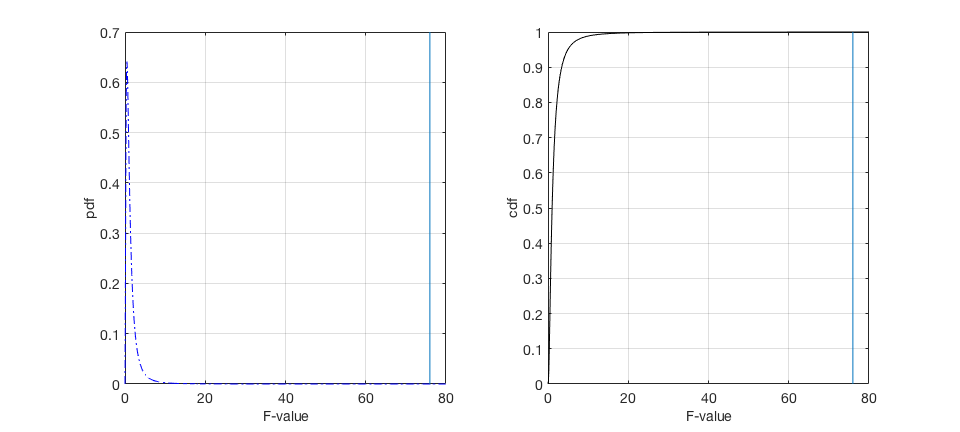
A distribution with degrees of freedom is the distribution of the sum of the squares of independent standard normal random variables ().

x = [0:.1:20];

y1 = fpdf(x,nu1,nu2);

y2 = fcdf(x,nu1,nu2);

F\_statistics = figure;

set(gcf,'Unit','Normalized',…

'Position',[0.2 0.5 0.5 0.4])

subplot(1,2,1)

plot(x,y1,'Color','blu','LineStyle','-.')

xlabel('F-value');ylabel('pdf');grid on

ca = gca; line([F F],ca.YLim);

subplot(1,2,2)

plot(x,y2,'Color','black','LineStyle','-')

xlabel('F-value');ylabel('cdf');grid on

line([F F],[0 1]);

In general, we reject *H*0 if *F*\* is large or, equivalently, if its associated *p*-value is small. The *p*-value answers the question: "what is the probability that we’d get an *F\** value as large as we did, if the null hypothesis were true?" To obtain the *p*-value of the regression from *F-statistics* we find the complement to 1 of the *F*\* value in the *F* distribution function (*cdf*) with degrees of freedom and :

nu1 = m-1

nu2 = m-n

msr = ssr/(nu1-nu2)

F = ((sst-sse)/(nu1-nu2))/(sse/nu2)

F = msr/mse

p\_f = 1-fcdf(F,nu1,nu2)

p\_f = fcdf(F,nu1,nu2,'upper')

p\_f = 9.5045e-05

QR factorization.

In practice for very large systems of equations (thousands or milions of experimental points) a numerically much more stable algorithm for the *least squares* procedure is based on the **QR** factorization of the design matrix ***A*** into an orthogonal matrix ***Q*** (a matrix with orthogonal columns of unit length) and an upper triangular matrix ***R*** as: ***A = QR***. In this case the *normal equation*  reduces to a much simpler equation, because the product of an orthogonal matrix by its transpose, ***Q'Q* = *I***, is the *identity matrix* ***I***. It follows that the new normal equation is:

which is instantly solved by Gaussian elimination and back substitution as is upper triangular:

[Q,R] = qr(A,0) % 'Economy' QR factorization

A

Q\*R

Q'\*yvec\_n

u\_qr = R\Q'\*yvec\_n

Weighted least squares.

Very often multiple independent experimental observations are available for each element of the ***b*** vector. In that case for each element we can determine both the mean and the standard deviation of the experimental observations. In particular, the ***σ*** of each determination can be used as a *weight* to give different importance to the different points. Weights can be introduced as a diagonal matrix ***W*** that multiplies each side of the matrix equation:

**)**

as a consequence the normal equation becomes:

and taking

Clearly the weights depend on the reliability of ***b*** and if each ***bi*** has a standard deviation ***σi***, a ***bi*** with larger standard deviation should have smaller weight. Thus, the best weights are given by ***wi*** = 1**/*σi*** and contains 1**/*σi*2** (inverse variances) on the diagonal. In this case, it is assumed that the *observations errors* are independent from each other (all off-diagonal terms are 0*)*. The new *weigthed least squares* (**WLS**) solution is:

which is the same as finding the minimum of:

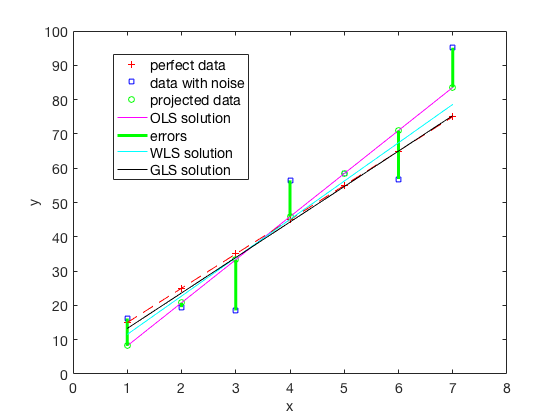
under the *constraint* of orthogonality between the column space of and . In fact, we can rearrange the normal equations as:

Notice that further simplification of is not possible because the rectangular matrix ***A*** has no inverse. We also notice that , the function minimed, corresponds to the traditional definition of :

Using the ***QR*** factorization, we derive:

**)** :

We can evaluate the effect of a *weight* matrix containing the inverse variances using the same data from the previous example. In this case we take only the diagonal from the covariance matrix:



W = 1./sqrt(diag(V));

W = diag(W)

% C = W'\*W;

C = diag(1./diag(V));

b = yvec\_n

u\_wls = (A'\*C\*A)\A'\*C\*b

A\_w = W\*A

b\_w = W\*b

[Q,R] = qr(A\_w,0)

u\_wls\_qr = R\Q'\*b\_w

u\_ols = A\b

yvec\_wls = u\_wls(1) + u\_wls(2)\*xvec;

figure(lsq\_plot)

h(13) = plot(xvec,yvec\_wls,'-c');

legend(h([1:5 13]), {'perfect data',…

'data with noise','projected data',...

'OLS solution','errors','WLS solution'},'Location','best')

The **WLS** accounts for the presence of gaussian noise in the experimental data. Also in this case, common criteria to evaluate the fit quality are:

*squared error*

*mean squared error* )

*sigma*

*Rsquared*

*adjusted Rsquared*

Finally, if the *error terms* (the *noise*) in ***b*** are not independent from each other, and the covariance matrix ***V*** of the errors is available, then we take **,** and the new *generalized least squares* (**GLS**) solution is:

C = inv(V);

u\_gls = (A'\*C\*A)\A'\*C\*b

yvec\_gls = u\_gls(1) + u\_gls(2)\*xvec;

figure(lsq\_plot)

h(14) = plot(xvec,yvec\_gls,'-k');

legend(h([1:5 13:14]), {'perfect data','data with noise','projected data', ...

'OLS solution','errors','WLS solution','GLS solution'},'Location','best')

with the same criteria used to estimate the quality of the **WLS** used also for the **GLS** fit.

For the **WLS** and **GLS** cases the *squared error*, , is the same as the *chi squared*, , the function minimized in the regression, and the *mean squared error*, , is the same as the *reduced chi squared*, . In these cases, the following rule of thumb (valid only when the variance of the measurement error is known *a priori* rather than estimated from the data) is often suggested for the interpretation of the *mean square error*, :

1. indicates a poor model fit.
2. indicates that the fit has not fully captured the data (or that the error variance has been underestimated).
3. indicates that the extent of the match between observations and estimates is in accord with the error variance.
4. indicates that the model is 'over-fitting' the data: either the model is improperly fitting noise, or the error variance has been overestimated.

Notice that, given:

regardless of the choice for ***C*** for**OLS***,***WLS**, or**GLS**, . Based on this observation we can derive the covariance matrix for the *output* error vector in our estimate of the optimal , corresponding to the *input* error in our observations. In fact, since and we derive:

The *covariance* matrix ***S*** of is the *expectation value* (the average value, typically represented with the symbol ***E***) for the product of the column vector times its transpose. Here is the *true* value for *intercept* and *slope* we have used to build the synthetic data, and is the *least squares* determination of for each observation.

Likewise, the *covariance* matrix ***V*** for the observations is the expectation value of the product of the column vector of observation errors times its transpose. Here is the theoretical error, the difference between the noise added of each observation and the perfect data before the addition of any noise:

V3 = zeros(m,m,nsamples);

S3 = zeros(n,n,nsamples);

C = inv(V);

u\_vec = [5 10]';

for i = 1:nsamples

yvec\_n = yvec\_m(:,i);

u\_ls = (A'\*C\*A)\A'\*C\*yvec\_n;

error\_u = u\_ls - u\_vec;

error\_b = yvec\_n - yvec;

S3(:,:,i) = error\_u\*error\_u';

V3(:,:,i) = error\_b\*error\_b';

end

E\_S3 = mean(S3,3)

E\_V3 = mean(V3,3)

Substituting for ***L*** and recalling that and thus we obtain:

The matrix is called the *information matrix* (or more generally the *precision matrix*)*.* It goes up as ***V*** goes down due to better observations, and also goes up as the experiment continues because new rows are added to ***A***. Notice that ***S***  does not depend on a specific ***b*** vector, but only on ***A*** and ***V***, with the latter (the covariance matrix of the observation errors) depending only on the experimental setup. Knowledge of ***S*** tells, in advance of any particular observation ***b***, how good the experiment should be. Since ***S*** is the covariance matrix of , from it we easily derive an estimate for the the *standard errors* (***se***) for the coefficients (that is, the *estimated standard deviation* ***σ*** of the distribution of each coefficient) as:

The 95% confidence intervals is obtained by subtracting or adding 1.96\***σ** to the 'expected' value .

We also notice that since , the projection ***p*** of ***b*** onto **C**(***A***) is:

This matrix is also *idempotent*, but not *symmetric*, which implies that is an *oblique* (rather than *orthogonal*) projection of ***b*** onto **C**(***A***). For example, the vector ***eGLS*** of **GLS** residuals is:

where:

is also*idempotent*, but not *symmetric*. It is important to notice here that the vector of **GLS** residuals is not orthogonal to the *column space* of , In fact, we had already derived rearranging the normal equation:

Thus, is in the *null space* of or *left nullspace* of , and thus it is orthogonal to the *column space* of .

MATLAB provides a sophisticated function, *lscov*,to carry out both ordinary least-squares (**OLS**) and weighted least squares (**WLS**), most often using as weight vector the inverse variances of the observations in ***b***. The weights can also be arranged as the diagonal of a matrix ***C***,with all diagonal terms typically different from each other (*heteroskedasticity condition*) and all off-diagonal terms equal to 0 (indicating the error terms in ***b*** are independent).

Finally, if the *error terms* in ***b*** are not independent from each other, *lscov* can use the observations *covariance matrix* ***V***, and to carry out a generalized least squares, **GLS**). Notice that if ***V*** is not provided it is assumed to be the *m x m* identity matrix: this is a special case of the *homoskedasticity condition*, which occurs when all diagonal term of ***C*** are (approximately) equal, regardless of whether the errors are correlated or not.

If the covariance matrix of the observations vector ***b*** is known only up to a scale factor (for example because it was derived from the instrument properties without any knowledge of the actual experimental observations), *lscov* uses the as the scale factor:

in which case we have:

(estimated covariance matrix of )

*lscov* by default applies this scale factor in all cases. If the covariance matrix is derived from the actual experimental observations, then the ***S*** matrix must be rescaled by dividing it by, and the standard errors of the regression parameters must be recalculated from the rescaled ***S***.

The following summarizes MATLAB *lscov* syntax for the different forms of least squares. This time we take the observations vector ***b*** to be the mean of 10 experiments:

B = (yvec\_m(:,randi(nsamples,10,1)))'

b = mean(B)'

I = eye(size(b,1))

Parameters for OLS, WLS,GLS

OLS

C\_ols = I;

WLS

w\_wls = var(B,1)

V\_wls = diag(var(B,1))

C\_wls = inv(V\_wls)

GLS

V\_gls = cov(B,1)

C\_gls = inv(V\_gls)

Ordinary Least Squares (**OLS**) - only the mean of multiple observations is used. Covariance matrix is assumed to be ***I*** (all errors are uncorrelated with unit variance) or ***σ*2*I*** (all errors are uncorrelated with constant variance ***σ*2**, the default in *lscov*).

# [u\_ols,se\_u\_ols,scale\_ols,S\_ols] = lscov(A,b,I) % also just lscov(A,b)

S\_ols\_corr = S\_ols/scale\_ols % correction to remove scaling

se\_u\_ols\_corr = sqrt(diag(S\_ols\_corr)) % correction to remove scaling

conf\_95\_ols = [u\_ols-1.96\*se\_u\_ols u\_ols+1.96\*se\_u\_ols]

Weighted Least Squares (**WLS**) - mean data is weighted according to its inverse variance

[u\_wls,se\_u\_wls,scale\_wls,S\_wls] = lscov(A,b,V\_wls)

S\_wls\_corr = S\_wls/scale\_wls % correction to remove scaling

se\_u\_wls\_corr = sqrt(diag(S\_wls\_corr)) % correction to remove scaling

conf\_95\_wls = [u\_wls-1.96\*se\_u\_wls u\_wls+1.96\*se\_u\_wls]

Generalized Least Squares (**GLS**) - mean data is weighted according to its covariance matrix

[u\_gls,se\_u\_gls,scale\_gls,S\_gls] = lscov(A,b,V\_gls)

S\_gls\_corr = S\_gls/scale\_gls % correction to remove scaling

se\_u\_gls\_corr = sqrt(diag(S\_gls\_corr)) % correction to remove scaling

conf\_95\_gls = [u\_gls-1.96\*se\_u\_gls u\_gls+1.96\*se\_u\_gls]

Plotting all together

ols\_wls\_gls\_plot\_1 = figure

xvec = A(:,2);

Get handles for each variable plotted

h1(1:12) = plot(xvec,B,'xk',xvec,b,'or',xvec,yvec,'sb')

hold on

yvec\_ols = u\_ols(1) + u\_ols(2)\*[0;xvec;8];

yvec\_wls = u\_wls(1) + u\_wls(2)\*[0;xvec;8];

yvec\_gls = u\_gls(1) + u\_gls(2)\*[0;xvec;8];

h1(13:15) = plot([0;xvec;8],yvec\_ols,'-m',[0;xvec;8],yvec\_wls,'-r',...

[0;xvec;8],yvec\_gls,'-b');

Select only some handles for the legend

legend(h1(10:15), 'multiple observations','mean data',...

'real data','OLS solution',...

'WLS solution ','GLS solution ','Location','best')

xlabel('x'),ylabel('y')

Observed residuals

r\_ols = yvec\_ols(2:8) - b

o\_mse\_ols = (r\_ols'\*r\_ols)/(m-n)

r\_wls = yvec\_wls(2:8) - b

o\_mse\_wls = (r\_wls'\*r\_wls)/(m-n)

r\_gls = yvec\_gls(2:8) - b

o\_mse\_gls = (r\_gls'\*r\_gls)/(m-n)

The exact same result is obtained just using standard MATLAB code in a more explicit way:

Ordinary Least Squares (**OLS**) using standard MATLAB syntax

u\_ols\_m = (A'\*C\_ols\*A)\A'\*C\_ols\*b

% e\_ols\_m = (I - A/(A'\*C\_ols\*A)\*A'\*C\_ols)\*b % error by projection

e\_ols\_m = b - A\*u\_ols\_m % error direct

% A'\*e\_ols\_m

% A'\*C\_ols\*e\_ols\_m

% All equivalent

sse = e\_ols\_m'\*C\_ols\*e\_ols\_m

sse = b'\*(C\_ols-C\_ols\*A/(A'\*C\_ols\*A)\*A'\*C\_ols)\*b

mse = sse/(m-n)

sigma = sqrt(mse)

b\_mean = b - mean(b);

sst = b\_mean'\*b\_mean;

rsquare = 1-sse/sst

S = inv(A'\*C\_ols\*A)

se\_u\_ols\_m = sqrt(diag(S))

conf\_95\_ols\_m = [u\_ols\_m-1.96\*se\_u\_ols\_m u\_ols\_m+1.96\*se\_u\_ols\_m]

scaled\_S = mse\*S

scaled\_se\_u\_ols\_m = sqrt(diag(scaled\_S))

scaled\_conf\_95\_ols\_m = [u\_ols\_m-1.96\*scaled\_se\_u\_ols\_m ...

u\_ols\_m+1.96\*scaled\_se\_u\_ols\_m]

Weighted Least Squares (**WLS**) using standard MATLAB syntax

u\_wls\_m = (A'\*C\_wls\*A)\A'\*C\_wls\*b

% Using QR

W = sqrt(C\_wls);

A\_w = W\*A;

b\_w = W\*b;

[Q,R] = qr(A\_w,0);

u\_wls\_qr = R\Q'\*b\_w

% same as

[Q,R] = qr(A,0);

s = (Q'\*C\_wls\*Q)\Q'\*C\_wls\*b;

t = Q\*s;

u\_wls\_qr = R\Q'\*t

% e\_wls\_m = (I - A/(A'\*C\_wls\*A)\*A'\*C\_wls)\*b % error by projection

e\_wls\_m = b - A\*u\_wls\_m % error direct

% A'\*e\_wls\_m

% A'\*C\_wls\*e\_wls\_m

% All equivalent

sse = e\_wls\_m'\*C\_wls\*e\_wls\_m

sse = b'\*(C\_wls-C\_wls\*A/(A'\*C\_wls\*A)\*A'\*C\_wls)\*b

mse = sse/(m-n)

sigma = sqrt(mse)

b\_mean = b - mean(b);

sst = b\_mean'\*b\_mean;

rsquare = 1-sse/sst

S = inv(A'\*C\_wls\*A)

se\_u\_wls\_m = sqrt(diag(S))

conf\_95\_wls\_m = [u\_wls\_m-1.96\*se\_u\_wls\_m u\_wls\_m+1.96\*se\_u\_wls\_m]

scaled\_S = mse\*S

scaled\_se\_u\_wls\_m = sqrt(diag(scaled\_S))

scaled\_conf\_95\_wls\_m = [u\_wls\_m-1.96\*scaled\_se\_u\_wls\_m ...

u\_wls\_m+1.96\*scaled\_se\_u\_wls\_m]

Generalized Least Squares (**GLS**) using standard MATLAB syntax

u\_gls\_m = (A'\*C\_gls\*A)\A'\*C\_gls\*b

% Using QR

[Q,R] = qr(A,0);

s = (Q'\*C\_gls\*Q)\Q'\*C\_gls\*b;

t = Q\*s;

u\_gls\_qr = R\Q'\*t

% e\_gls\_m = (I - A/(A'\*C\_gls\*A)\*A'\*C\_gls)\*b % error by projection

e\_gls\_m = b - A\*u\_gls\_m % error direct

% A'\*e\_gls\_m

% A'\*C\_gls\*e\_gls\_m

% All equivalent

sse = e\_gls\_m'\*C\_gls\*e\_gls\_m

sse = b'\*(C\_gls-C\_gls\*A/(A'\*C\_gls\*A)\*A'\*C\_gls)\*b

mse = sse/(m-n)

sigma = sqrt(mse)

b\_mean = b - mean(b);

sst = b\_mean'\*b\_mean;

rsquare = 1-sse/sst

S = inv(A'\*C\_gls\*A)

se\_u\_gls\_m = sqrt(diag(S))

conf\_95\_gls\_m = [u\_gls\_m-1.96\*se\_u\_gls\_m u\_gls\_m+1.96\*se\_u\_gls\_m]

scale = b'\*(C\_gls - C\_gls\*A/(A'\*C\_gls\*A)\*A'\*C\_gls)\*b/(m-n)

scaled\_S = scale\*S

scaled\_se\_u\_gls\_m = sqrt(diag(scaled\_S))

scaled\_conf\_95\_gls\_m = [u\_gls\_m-1.96\*scaled\_se\_u\_gls\_m ...

u\_gls\_m+1.96\*scaled\_se\_u\_gls\_m]

Plotting all together

ols\_m\_wls\_m\_gls\_m\_plot = figure

xvec = A(:,2);

Get handles for each variable plotted

h1(1:12) = plot(xvec,B,'xk',xvec,b,'or',xvec,yvec,'sb')

hold on

yvec\_ols\_m = u\_ols\_m(1) + u\_ols\_m(2)\*[0;xvec;8];

yvec\_wls\_m = u\_wls\_m(1) + u\_wls\_m(2)\*[0;xvec;8];

yvec\_gls\_m = u\_gls\_m(1) + u\_gls\_m(2)\*[0;xvec;8];

h1(13:15) = plot([0;xvec;8],yvec\_ols\_m,'-m',[0;xvec;8],yvec\_wls\_m,'-r',...

[0;xvec;8],yvec\_gls\_m,'-b');

Select only some handles for the legend

legend(h1(10:15), 'multiple observations','mean data',...

'real data','OLS solution',...

'WLS solution ','GLS solution ','Location','best')

xlabel('x'),ylabel('y')

Observed residuals

r\_ols\_m = yvec\_ols\_m(2:8) - b

o\_mse\_ols\_m = (r\_ols\_m'\*r\_ols\_m)/(m-n)

r\_wls\_m = yvec\_wls\_m(2:8) - b

o\_mse\_wls\_m = (r\_wls\_m'\*r\_wls\_m)/(m-n)

r\_gls\_m = yvec\_gls\_m(2:8) - b

o\_mse\_gls\_m = (r\_gls\_m'\*r\_gls\_m)/(m-n)

In most cases application of the *normal equation* as **OLS** or **WLS** provides the BLUE of the regression parameters and the fit with the smallest residual. However, if the covariance matrix is known, the **GLS** solution may be more accurate ('closer to the truth') in the presence of significant *heteroskedasticity* and/or correlation between the observation errors.

*lscov* estimate of the standard deviations of the regression parameters by scaling the covariance matrix ***S*** is usually preferable to the estimate using an unscaled covariance matrix. However, in all cases the mean square error, *mse =* ***σ*2**, is better derived outside *lscov* by applying the traditional formula.

Key concepts:

1. The general LS solution for is:

where is the identity matrix in ***OLS***, a diagonal matrix of inverse variances in ***WLS***, and the inverse of the covariance matrix in ***GLS***.

1. The covariance matrix of is:

where ), and is the *precision* matrix; is the error, is the number of points in the vector, is the number of unknown in , and is x .

1. The square root of the diagonal of gives the standard deviation of the elements of , which is usually called the *standard error*, :
2. The error is always in the *left null space* of , regardless of what is.
3. The following rule applies when assembling a matrix ***D*** of 'nobs' observations of a random column vector of *m* variables. We can collect the data working by rows (a) or by columns (b):
4. if we collect in each experiment all the observations of a single variable, ***D*** can only be used to derive the variance of each variable.
5. if we collect in each experiment a single observation of all the variables, ***D*** can be used to derive a covariance matrix of all the variables.

**SPECIAL TOPIC: *Information* and *precision* matrix**

In this chapter we have shown that the covariance matrix ***S*** of the **GLS**estimator is:

where is the *information matrix.* It is easy to derive the **OLS** estimatorfrom the **GLS** estimator because in ordinary least squares we make the assumption that all the errors are uncorrelated with constant variance ***σ*2**, and thus ***V*** *=* ***σ*2*I*** and ***C*** *=**1/****σ*2*I***.

Since all the elements on the diagonal are the same, we can bring the matrix ***C*** outof of the inverse obtaining:

where is the *information matrix* for the **OLS** case. In the next chapter we will learn that the covariance matrix of the **OLS** estimator inside each iteration of *non-linear least squares* is:

where is the *precision matrix* and is the *Jacobian*, the matrix of 1st derivatives of the residual function ***g*** with respect to each of the variables in . Remarcably, we can easily show with a numerical example that also in the **OLS** estimator for ***Au*** = ***b***, the matrix is the *Jacobian* of ***b*** with respect to each of the variables in . Taking the example used previously:

xvec = [1:7]'; int = 5; slope = 10; A = [ones(7,1) xvec]

yvec = int + slope\*xvec;

lsq\_plot = figure; plot(xvec,yvec,'+r');

Here we calculate the *partial derivatives* of ***b*** with respect to each element in ***u***.

du = eps^(1/3);

u1 = int; u2 = slope

u1\_plus = u1+du; u2\_plus = u2+du;

yvec1\_plus = u1\_plus + u2\*xvec; yvec2\_plus = u1 + u2\_plus\*xvec;

u1\_minus = u1-du; u2\_minus = u2-du;

yvec1\_minus = u1\_minus + u2\*xvec; yvec2\_minus = u1 + u2\_minus\*xvec;

Here we assemble the *Jacobian*.

J = [(yvec1\_plus-yvec1\_minus)/(2\*du) (yvec2\_plus-yvec2\_minus)/(2\*du)]

This numerical result ( = ) proves the equivalence between *information* and *precision* matrix in the two applications of linear least squares.

**SPECIAL TOPIC: Multiple linear regression and regularization techniques**

The same principles we have applied in cases of linear least-squares to derive the regression parameters of *response data* to a single class of *predictors* (the independent variables, can be used when there is more than one class of predictors. We have seen how a *multiple linear regression* (MLR) model relates a dependent variable (*response*) to *p* independent variables (*predictors*, aka *explanatory variables* or *regressors*) , according to:

where we assume that the have a normal distribution with mean 0 and constant variance.

We can’t stress enough the fact that the word "linear" refers to the fact that the model is *linear* in the parameters, . This simply means that each parameter multiplies an ***x*** variable, and the regression function is a sum of these "parameter times ***x*** variable" terms. Each ***x*** variable can be a predictor variable or a transformation of predictor variables (such as the power of a predictor variable or two predictor variables multiplied together), allowing a multiple linear regression model to represent non-linear relationships between the response variable and the predictor variables.

In matrix notation an MLR model is represented as:

response design matrix slope error

vector vector vector

The regression coefficients (the slope vector) are determined as usual by employing the normal equation:

and the residual is:

with:

and the *standard error* (se):

For example, suppose we want to determine the relationship between the fuel efficiency of a group of cars to two types of predictors, the car weight and the car horse power:

load carsmall

x1 = Weight;

x2 = Horsepower;

y = MPG;

Use only weight (X1):

X1 = [ones(size(x1)) x1];

Here we use MATLAB '*regress*' function:

b = regress(y,X1)

Here we use standard least squares:

ind = ~isnan(y);

b = X1(ind,:)\y(ind)

Use both weight (X1) and power (X2):

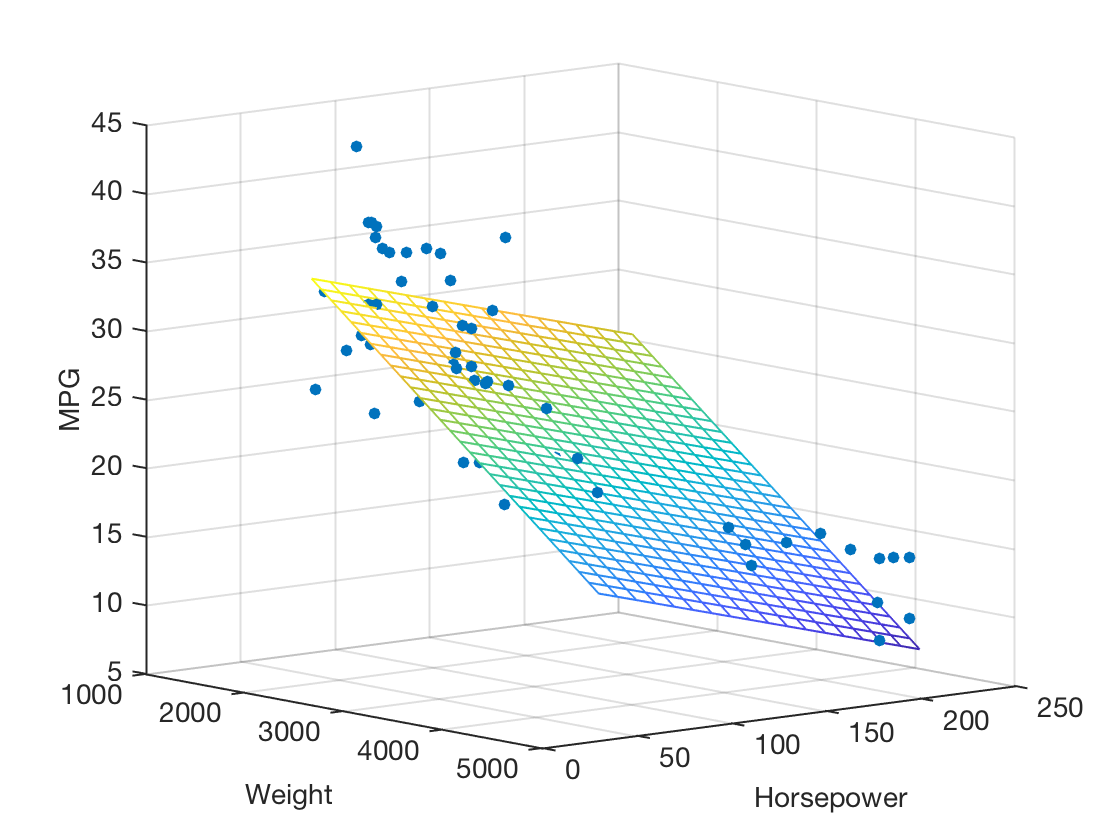
X12 = [ones(size(x1)) x1 x2];

b = regress(y,X12)

ind1 = ~isnan(y);ind2 = ~isnan(x2);

ind = logical(ind1.\*ind2)

b = X12(ind,:)\y(ind)

Since there are two predictors, we have a least-squares *plane* instead of a least-squares *line*:

scatter3(x1,x2,y,'filled');hold on

x1fit = min(x1):100:max(x1);

x2fit = min(x2):10:max(x2);

[X1FIT,X2FIT] = meshgrid(x1fit,x2fit);

YFIT = b(1) + b(2)\*X1FIT + b(3)\*X2FIT;

mesh(X1FIT,X2FIT,YFIT)

xlabel('Weight')

ylabel('Horsepower')

zlabel('MPG')

view(50,10)

Use both weight (X1) and power (X2) plus an interaction term:

X12i = [ones(size(x1)) x1 x2 x1.\*x2];

b = regress(y,X12i)

ind1 = ~isnan(y);ind2 = ~isnan(x2);

ind = logical(ind1.\*ind2)

b = X12i(ind,:)\y(ind)

The addition of an interaction term between the two predictors produces a least-squares curved surface and a better fit:

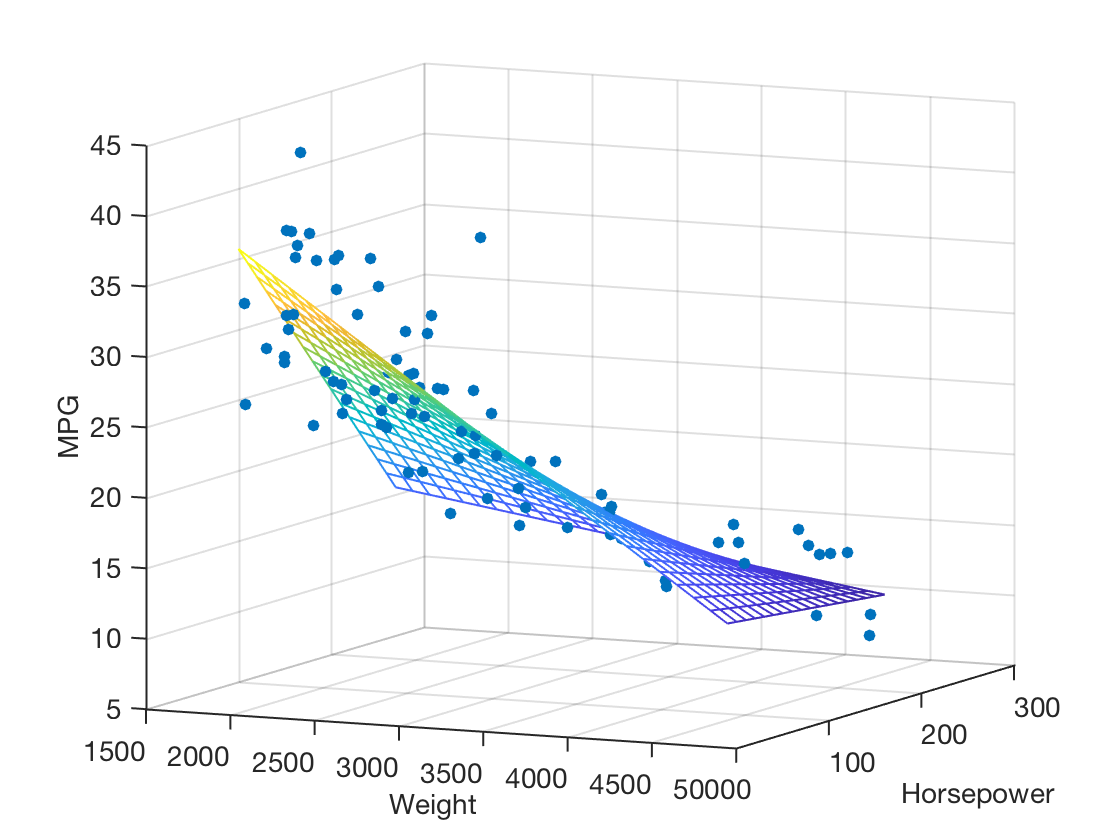
scatter3(x1,x2,y,'filled'); hold on

x1fit = min(x1):100:max(x1);

x2fit = min(x2):10:max(x2);

[X1FIT,X2FIT] = meshgrid(x1fit,x2fit);

YFIT = b(1) + b(2)\*X1FIT + ...

b(3)\*X2FIT + b(4)\*X1FIT.\*X2FIT;

mesh(X1FIT,X2FIT,YFIT)

xlabel('Weight')

ylabel('Horsepower')

zlabel('MPG')

view(50,10)

We have seen how the significance of each regressor can be derived from *t-*statistic:

number of observations

m = sum(ind)

number of parameters

n = length(b)

nu value

nu = m-n

e = y(ind,:) - X12i(ind,:)\*b;

mse = (e'\*e)/nu

standard error

se = sqrt(diag(mse\*inv(X12i(ind,:)'\*X12i(ind,:))))

student’s *t*

t = b./se

as we find the complement to 1 of the value of the Student's *t* cumulative distribution function (*cdf*) for each *t* value based on the known degrees of freedom :

p values

p\_t = 1-tcdf(t,nu)

p\_t = tcdf(t,nu,'upper')

p\_t\_2 = 2\*tcdf(abs(t),nu,’upper’) % for a 2-tail *p-value*

We have also seen that we can test for the *null hypothesis* ‘‘ (*reduced model* having only the intercept) against the *alternative hypothesis* ‘’ (*full model*) using *F-statistics*. In general, we reject the null hypothesisif *F*\* is large or if its associated *p*-value is small. To obtain the *p*-value of the regression we find the complement to 1 of the *F*\* value in the *F* distribution function (*cdf*) with numerator degree of freedom and denominator degrees of freedom:

sse = e'\*e; sst = (y(ind,:) - mean(y(ind,:)))'\*(y(ind,:) - mean(y(ind,:))); ssr = sst-sse;

r2 = 1-sse/sst; r2 = ssr/sst

adj\_r2 = (sse/(m-n))/(sst/m-1)

nu1 = m-1

nu2 = m-n

msr = ssr/(nu1-nu2)

F = ((sst-sse)/(nu1-nu2))/(sse/nu2)

F = msr/mse

p\_f = 1-fcdf(F,nu1,nu2)

p\_f = fcdf(F,nu1,nu2,'upper')

While SST is a constant term, regardless of which predictors are included in the regression model, SSE (and thus SSR) depends on the included predictors. This observation allows us to carry out a multiple linear regression in a *stepwise* manner.

*Stepwise regression* is a systematic method for adding and removing terms from a multilinear model based on their statistical significance in a regression. The method begins with an initial model and then compares the explanatory power of incrementally larger and smaller models. At each step, the *p*-value of a *partial* *F*-*statistic* is computed to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

As an example, consider adding to the previous example of multiple linear regression also the ‘Acceleration’ variable as a predictor:

x1 = Weight;

x2 = Horsepower;

x3 = Acceleration;

y = MPG;

X123 = [x1 x2 x3];

Here we use the function x2fx to derive a design matrix containing all possible interactions:

D = x2fx(X123,'interaction')

Predictors = {'x1','x2','x3','x1x2','x1x3','x2x3'}

[nobs,npred] = size(D);

Here we select only the three original predictors X1, X2, and X3 plus a single interaction X1\*X2:

D = D(:,1:5)

[b,bint,r,rint,stats] = regress(y,D)

ind1 = ~isnan(y)

ind2 = ~isnan(x2)

ind3 = ~isnan(x3)

ind = logical(ind1.\*ind2.\*ind3)

b = D(ind,:)\y(ind)

Standard multiple regression *t* statistics

number of observations

m = sum(ind)

number of parameters

n = length(b)

nu value

nu = m-n

e = y(ind,:) - D(ind,:)\*b;

mse = (e'\*e)/nu

standard error

se = sqrt(diag(mse\*inv(D(ind,:)'\*D(ind,:))))

student’s *t*

t = b./se

*p* values

p\_t = 1-tcdf(t,nu)

p\_t = tcdf(t,nu,'upper')

p\_t\_2 = 2\*tcdf(abs(t),nu,'upper')   % for a 2-tail p-value

Table of response and predictors:

T = table(y(ind),x1(ind),x2(ind),x3(ind),x1(ind).\*x2(ind),'VariableNames',...

    {'MPG' 'Weight' 'Horsepower' 'Acceleration' 'WxH'})

Here we carry out a *stepwise* fit. We use again the function x2fx to derive a design matrix containing all possible interactions:

D = x2fx(X123,'interaction')

[nobs,npred] = size(D);

D(:,1) = []

[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(D,y)

Initial columns included: none

Step 1, added column 1, p=3.24054e-28

Step 2, added column 2, p=0.02686

Step 3, added column 4, p=0.00402917

Final columns included: 1 2 4

'Coeff' 'Std.Err.' 'Status' 'P'

[ -0.0102] [ 0.0016] 'In' [6.2024e-09]

[ -0.1882] [ 0.0527] 'In' [5.6978e-04]

[ -0.1966] [ 0.1938] 'Out' [ 0.3131]

[ 3.8495e-05] [1.3038e-05] 'In' [ 0.0040]

[-6.7990e-05] [6.0878e-05] 'Out' [ 0.2671]

[ -0.0020] [ 0.0016] 'Out' [ 0.2156]

b\_in = [stats.intercept;b(inmodel)];

m = sum(ind)

n = length(b\_in)

p\_vec = [ones(m,1) D(ind,inmodel)]\*b\_in;

e = y(ind) - p\_vec;

SSE = e'\*e; SST = (y(ind) - mean(y(ind)))'\*(y(ind) - mean(y(ind)));

SSR = SST - SSE

r2 = 1-SSE/SST

r2 = SSR/SST

adj\_r2 = (SSE/(m-n))/(SST/m-1)

r2 = 0.7742

adj\_r2 = 0.2397

However, determination of the regression coefficient estimates for this linear model rely on the independence of the model terms, and we can easily recognize that there is significant correlation between the two predictors *Weight,* and *Horsepower*:

corr(D(ind,[2:5])) ans = 1.0000 0.8656 0.9256 -0.4642

0.8656 1.0000 0.9794 -0.6836

0.9256 0.9794 1.0000 -0.6030

-0.4642 -0.6836 -0.6030 1.0000

and only these two predictors are significantly correlated to *MPG*, as shown by the stepwise regression.

corr([y(ind) D(ind,[2:5])]) ans = 1.0000 -0.8591 -0.8028 -0.8181 0.4631

-0.8591 1.0000 0.8656 0.9256 -0.4642

-0.8028 0.8656 1.0000 0.9794 -0.6836

-0.8181 0.9256 0.9794 1.0000 -0.6030

0.4631 -0.4642 -0.6836 -0.6030 1.0000

When terms are correlated and the columns of the design matrix **X** have an approximate linear dependence, the matrix  becomes close to singular. As a result, the least-squares estimate:

becomes highly sensitive to random errors in the observed response . This situation of *multicollinearity* can arise, for example, when data are collected without a well planned experimental design. There are several ways of addressing this problem, which essentially all work by *shrinking* the number of predictors, and/or selecting those with minimal redundancy between them. *Stepwise regression* is in fact one of these ways. Three other methods, *Ridge regression*, *lasso*, and *elastic net* are usually classified as *regularization techniques*.

In *Ridge regression* the regression coefficients are estimated using:

where *k* is a *ridge parameter* and ***I*** is the identity matrix. Small positive values of *k* improve the conditioning of the problem by inflating the eigenvalues of that are closer to 0 and thus reduce the variance of the regression coefficients. Usually different small values of *k* are tried in an increasing range [0:10-2], which progressively *shrink* the estimates until they converge to some values, or until the desired number of coefficients converge to 0.

Like *ridge regression*, *Lasso* includes a penalty term that *shrinks* the size of the estimated coefficients. For a given value of , lasso solves the problem:

As the penalty term increases, *lasso* sets more coefficients to zero, generating a model with fewer predictors. For this reason, *lasso* is usually considered a good alternative to *stepwise regression*. When calling *lasso* in MATLAB we typically set the parameter *DFmax* (which defines the maximum number of predictors we want to retain), and the parameter , which determines whether the function behaves as *ridge regression* () or as *lasso* (). For all intermediate values of the penalty term interpolates between the 1-norm and 2-norm of , and the regression is usually referred to as *elastic net*.

In the following we compare *stepwise* and *lasso* regression.

X123 = [x1 x2 x3]

Here we use the function x2fx to derive a design matrix containing all possible interactions:

D = x2fx(X123,'interaction')

[nobs,npred] = size(D);

D(:,1) = []

Here we use 'stepwise' regression:

[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(D,y)

Initial columns included: none

Step 1, added column 1, p=3.24054e-28

Step 2, added column 2, p=0.02686

Step 3, added column 4, p=0.00402917

Final columns included: 1 2 4

'Coeff' 'Std.Err.' 'Status' 'P'

[ -0.0102] [ 0.0016] 'In' [6.2024e-09]

[ -0.1882] [ 0.0527] 'In' [5.6978e-04]

[ -0.1966] [ 0.1938] 'Out' [ 0.3131]

[ 3.8495e-05] [1.3038e-05] 'In' [ 0.0040]

[-6.7990e-05] [6.0878e-05] 'Out' [ 0.2671]

[ -0.0020] [ 0.0016] 'Out' [ 0.2156]

b\_in = [stats.intercept;b(inmodel)];

number of observations

m = sum(ind)

number of parameters

n = length(b\_in)

p\_vec = [ones(m,1) D(ind,inmodel)]\*b\_in;

e = y(ind) - p\_vec;

SSE = e'\*e; SST = (y(ind) - mean(y(ind)))'\*(y(ind) - mean(y(ind)));

SSR = SST - SSE

r2 = 1-SSE/SST

r2 = SSR/SST

adj\_r2 = (SSE/(m-n))/(SST/m-1)

r2 = 0.7742

adj\_r2 = 0.2397

Here we use 'lasso' keeping the same number of predictors (DFmax = 3):

[b,FitInfo] = lasso(D,y,'PredictorNames',{'x1','x2','x3','x1x2','x1x3','x2x3'},'DFmax',3)

best\_predictors\_ind = find(b(:,1))

best\_predictors = Predictors(best\_predictors\_ind)

best\_predictors = 'x1' 'x2' 'x2x3'

D\_best = [ones(nobs,1) D]

p\_vec = D\_best(ind,:)\*[FitInfo.Intercept(1);b(:,1)];

e = y(ind) - p\_vec;

m = sum(ind)

n = sum(b(:,1)~=0)

SSE = e'\*e; SST = (y(ind) - mean(y(ind)))'\*(y(ind) - mean(y(ind)));

SSR = SST - SSE

r2 = 1-SSE/SST

r2 = SSR/SST

adj\_r2 = (SSE/(m-n))/(SST/m-1)

r2 = 0.7520

adj\_r2 = 0.2603

Here we use 'lasso' with 10-fold *cross-validation* keeping the same number of predictors (DFmax = 3):

[b,FitInfo] = lasso(D,y,'CV',10,'PredictorNames',{'x1','x2','x3','x1x2','x1x3','x2x3'},'DFmax',3)

minMSEModel = FitInfo.PredictorNames(b(:,FitInfo.IndexMinMSE)~=0)

minMSEModel = 'x1' 'x2' 'x2x3'

D\_best = [ones(nobs,1) D]

p\_vec = D\_best(ind,:)\*[FitInfo.Intercept(1);b(:,FitInfo.IndexMinMSE)];

e = y(ind) - p\_vec;

m = sum(ind)

n = sum(b(:,1)~=0)

SSE = e'\*e; SST = (y(ind) - mean(y(ind)))'\*(y(ind) - mean(y(ind)));

SSR = SST - SSE

r2 = 1-SSE/SST

r2 = SSR/SST

adj\_r2 = (SSE/(m-n))/(SST/m-1)

r2 = 0.7515

adj\_r2 = 0.2608

**PRACTICE**

**1.** We want to plot (on the Y axis) the % increase in the activity of a given enzyme in the presence of increasing concentration (on the X axis) of the allosteric modulator AM. The X and Y values to be plotted are:

xvec = [1 2 3 4 5 6 7 8 9];

yvec = [1 3 2.5 3.5 3 2.5 2.5 2 4]';

The experimental points are clearly very scattered, and we would like to plot, superimposed on the experimental points, also some 'smoother' analytical function that would represent our model of allosteric control. We decide to use a polynomial fit of the experimental data. In mathematics, a **polynomial** is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponentiation. They are typically, but not exclusively, used to represent *non-linear functions*. This means that if we treat a polynomial *pol* like a *function*, each *yi* point can be considered as the application of the polynomial function to the *xi* point, that is:

In our case, we have a total of 9 points, and we know that the higher the degree of the polynomial we use in the fit, the closer our model will be to the original points. Thus we want to choose a polynomial of lesser degree than the total number of points.

Hint: a fit of ***m*** points with a polynomial of degree ***n-1*** (with ***n < m***)is based on finding the ***n*** coefficients of the polynomial by solving the system of linear equations:

...

...

...

which can be written as a matrix equation:

*m* x n *n* x 1 *m* x 1

The *m* x *n* matrix is called the *Vandermonde* matrix. You can see that this matrix has full column rank (all columns are linearly independent) and if the number of points *m* is equal to the number of coefficients n(square matrix), that is if we choose to fit the *m* points with a polynomial of degree *m-1*, then the equation has an exact solution. However, if the number of points to fit is larger than the number of coefficients (*m* > n), the system is overdetermined and an exact solution does not exist.

Start by getting the Vandermonde matrix:

A = vander(xvec)

a. Write a program (optimally a function) that will calculate and plot a polynomial fit of a desired degree, and provide the fit parameters (i.e., sse, sigma, R-squared). Your program should give the same polynomial coeffcients as the MATLAB function 'polyfit':

pol\_coeff = polyfit(xvec,yvec,deg)

where deg (= *n - 1*) is the degree of the polynomial.

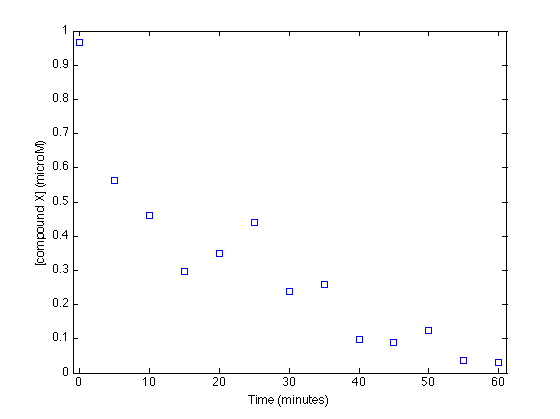
b. Explain the relationship between 'less than optimal' polynomial fit and the procedure of *least squares*.

**2.** We have injected a patient intravenously with 90 mg of antibiotic Y (MW = 300) for which we have developed a very sensitive spectrophotometric assay using just a few microliters of blood. After **1 hour**, we start measuring the concentration of the antibiotic in the plasma every 5 minutes and we obtain the following result with our assay:

t = [0 5 10 15 20 25 30 35 40 45 50 55 60]' % (times in minutes)

Y = [0.9669 0.5627 0.4608 0.2979 0.3493 0.4414 0.2387 0.2586 ...

0.0988 0.0896 0.1247 0.0378 0.03031]' % (microM)



We know that the antibiotic distributes within seconds to all the extracellular fluid in the body, is not metabolized, and is cleared exclusively through the kidney. Based on our result we suspect that the antibiotic clearance follows 1st order elimination kinetics (exponential decrease).

In this case we have a non-linear function of the type:

where ***Y0*** is the concentration of the antibiotic at ***t*** = 0 and ***k***is the decay constant*,* which cannot be solved with the type of *linear* *least square* (also called *'linear regression'*) we have learned. However, the problem can be transformed to a linear regression by taking the logarithm of both sides:

and fitting this new function to the data points(***ti***, *ln*(***Yi***)). Notice that in this case again the 1st column of ***A*** is all 1's, the 2nd column (the ***t*** vector)are the time points, *ln(****Y0****)* is the intercept, and ***k*** is the slope.

***A x b***

1. Is the elimination model plausible? What are the fit parameters and the *standard deviation* of the fit?

2. Is there any relationship between the ***σ*** and the error ***e*** in the least squares solution of the exponential model?

3. Knowing that antibiotic X is not transported inside the body cells, calculate from the elimination data the total volume of the extracellular fluids in the patient.

4. We have repeated the antibiotic injection 5 times in different days: thus instead of a single vector of concentrations we now have a matrix. In some cases (NaN) the antibiotic could not be detected.

Y\_mat = ...

[0.849,0.734,0.591,0.5,0.342,0.331,0.262,0.0484,0.0403,0.0311,0.0312,0.0318,0.051;

0.696,0.733,0.731,0.365,0.462,0.322,0.22,0.192,NaN,0.0969,0.243,0.0738,0.0539;

0.926,0.776,0.63,0.515,0.33,0.263,0.426, NaN,0.359,0.139,0.182, NaN, NaN;

0.972,0.821,0.439,0.52,0.246,0.293,0.289,0.207,0.244,0.206,0.0168,0.0897, NaN;

0.867,0.872,0.607,0.467,0.459,0.346,0.258,0.299,0.229,0.129,0.0129, NaN,0.169]

Use a *weighted least squares procedure* to obtain new values for the fit parameters and for the standard deviation of the exponential fit.

Hint: due to the absence of some points in the matrix (NaN, as 'not a number') use 'nanmean' and 'nanstd' to calculate the mean and standard deviation for each time point.