[1]:
```
# make figures better:
%matplotlib inline
import matplotlib
font = {'weight':'normal','size':22}
matplotlib.rc('font', **font)
matplotlib.rc('figure', figsize=(9.0, 6.0))
matplotlib.rc('xtick.major', pad=10) # xticks too close to border!

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')

import warnings
warnings.filterwarnings('ignore')
```

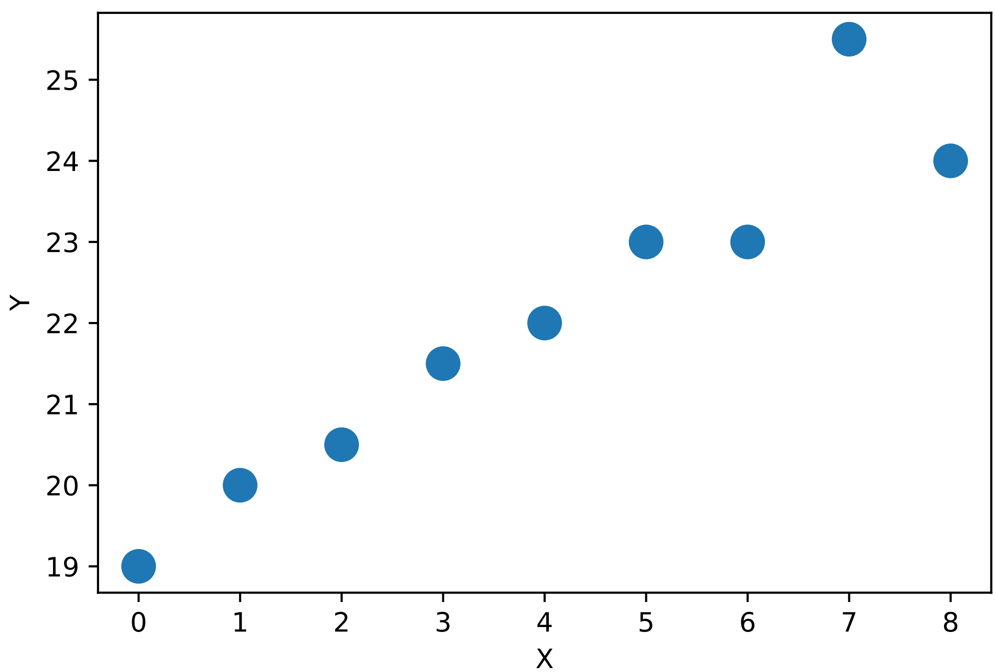
DS1 Lecture 12

James Bagrow, james.bagrow@uvm.edu, http://bagrow.com

Linear regression

Again

[2]:
```
# a few data points
X = [0,1,2,3,4,5,6,7,8]
Y = [19, 20, 20.5, 21.5, 22, 23, 23, 25.5, 24]

plt.plot(X,Y,'o', ms=12)
plt.xlabel("X"); plt.ylabel("Y");
```



Is there a **linear** relationship between the two?

- A constant change Δx in x leads to a constant change Δy in y, independent of x.

How to do linear regression:

- Fitting a straight line $y = f(x) = mx + b$ by estimating the parameters m (slope) and b (y-intercept) that minimize the **sum of squared errors**:

$$\sum_i (y_i - f(x_i))^2 = \sum_i (y_i - b - mx_i)^2 \tag{1}$$

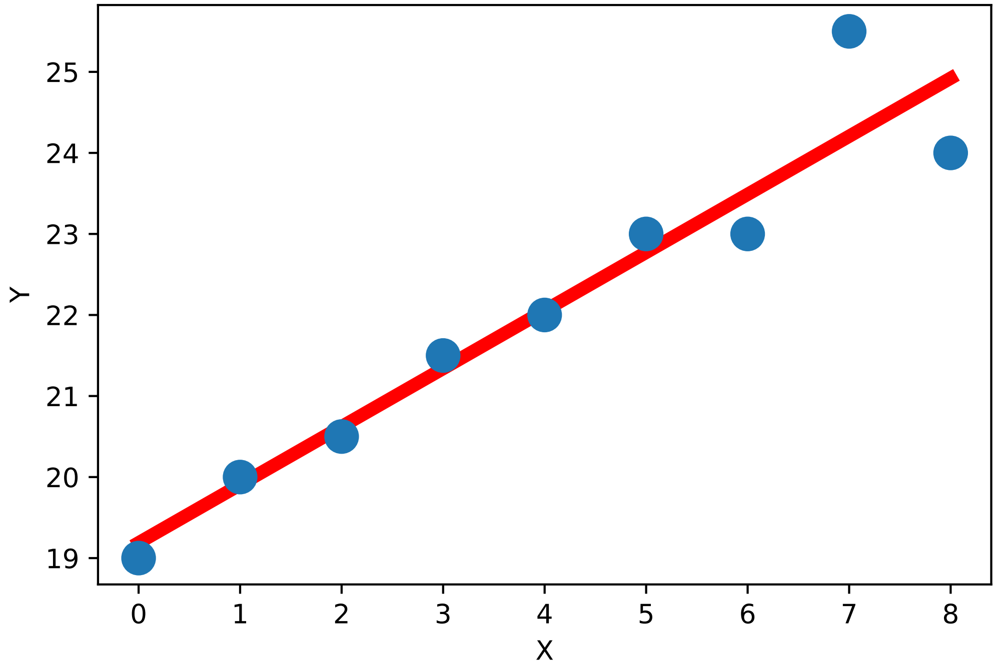
(This is actually called *simple* linear regression because there is only one X-variable.)

Common ways to do linear regression in Python:

- `polyfit` (from either `numpy` or `pylab`)
- `scipy.stats.linregress`
- `statsmodels.api.OLS`

[3]:
```
import scipy, scipy.stats

# let's fit a line:
slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(X,Y)
#                  ^^^^^^^  ^^^^^^^  ^^^^^^^ ?

print("    slope =", slope)
print("intercept =", intercept)


# Here's our function:
line = [ slope*xi + intercept for xi in X]

# plot it up
plt.plot(X,line,'r-', linewidth=5)
plt.plot(X,Y,'o', ms=12)
plt.xlabel("X"); plt.ylabel("Y");
```

```
    slope = 0.7166666666666667
intercept = 19.18888888888889
```



Question Is there really a relationship between x and y, or could the observed slope be **due to chance**?

Let's pretend for a second we don't know any statistics. Can we say this linear regression slope is *significant*?

Ans:

Monte Carlo Permutation Test

- Hypothesis: There is a significant relationship between X and Y variables
- Null (boring explanation): There is no relationship, X and Y are independent.

We can use the computer to break any link between X and Y in the data:

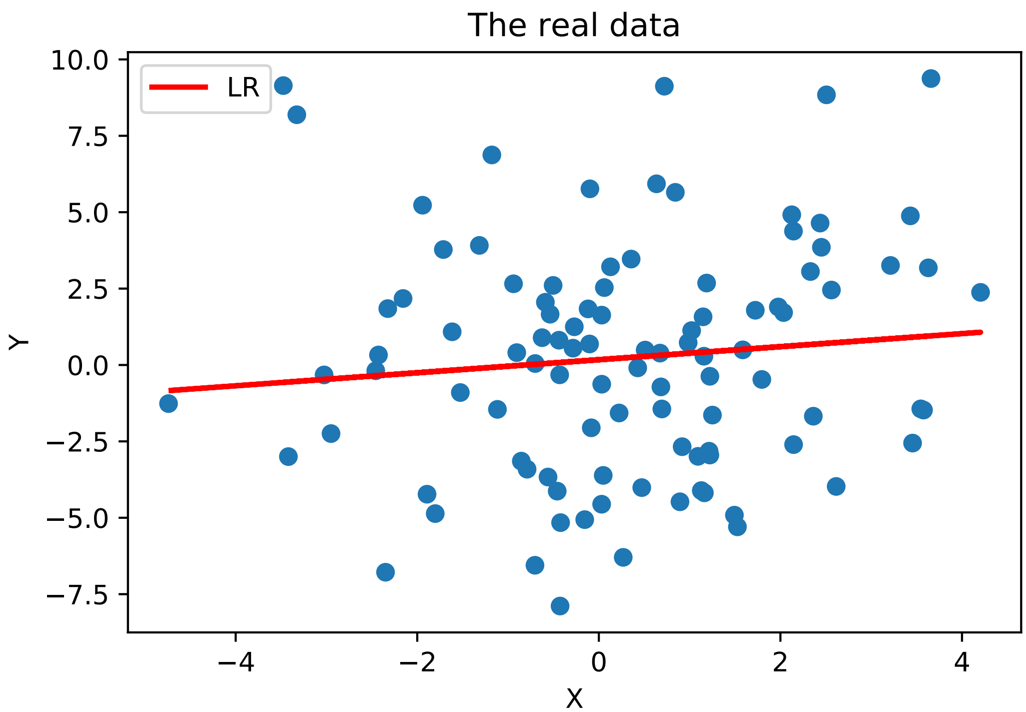
- How? Take the Y vector and **shuffle it** (permute it), mixing the Y_i data points across the X_i points at random. This makes the null hypothesis true!

First, some "real" data:

[4]:
```
# data:
np.random.seed(42) #"fix" the randomness, for class demo
x = 2 * np.random.randn(100) + 0.5
y = 4 * np.random.randn(100) + 0.5*x
#   ^ noisy!                     ^^^ real slope

# linear regression x and y
slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x,y)


plt.plot(x,y,'o', label=None)
plt.plot(x, slope*x+intercept, 'r-', lw=2, label="LR")
plt.xlabel("X"); plt.ylabel("Y")
plt.title("The real data")
```

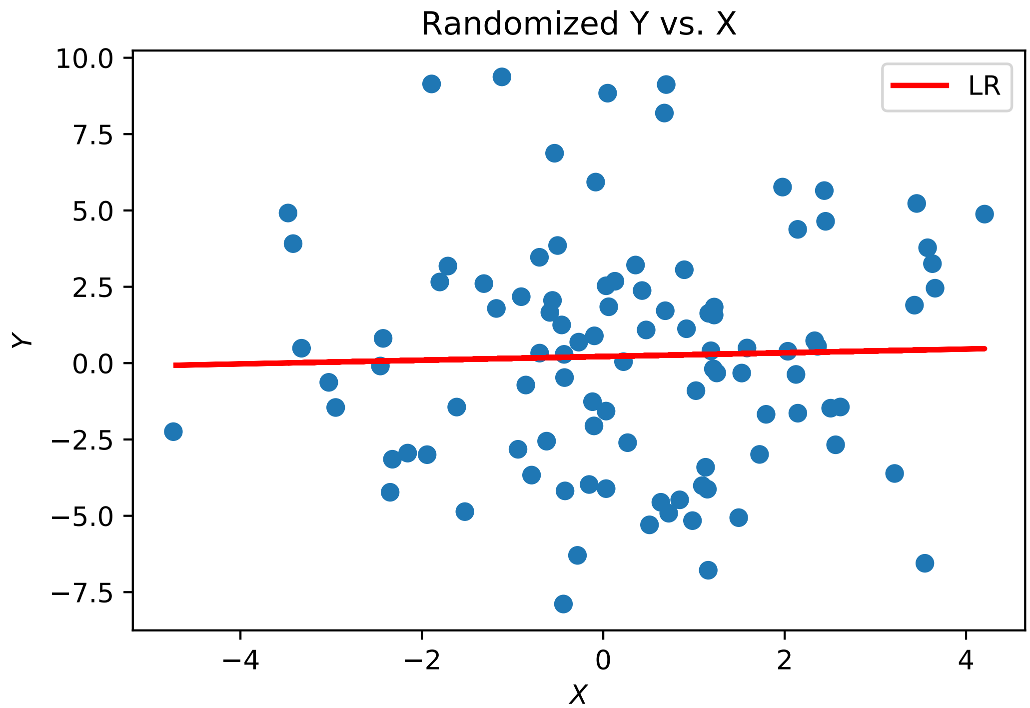
```
plt.legend()

print(slope, intercept)
print(r_value, p_value, std_err)
```

```
0.21348567945711308 0.17296847972731122
0.10207340256674752 0.3122351008567024 0.2101690672411271
```



Now, let's **force** there to be no relationship between x and y in the data:

[5]:
```
# shuffle a copy of y:
yr = y.copy()
np.random.shuffle(yr)

slopeR, interceptR, _, _, _ = scipy.stats.linregress(x,yr)

plt.title("Randomized Y vs. X")
plt.plot(x,yr,'o', label=None)
plt.plot(x, slopeR*x+interceptR, 'r-', lw=2, label="LR")
plt.xlabel("$X$")
plt.ylabel("$Y$")
plt.legend();
```



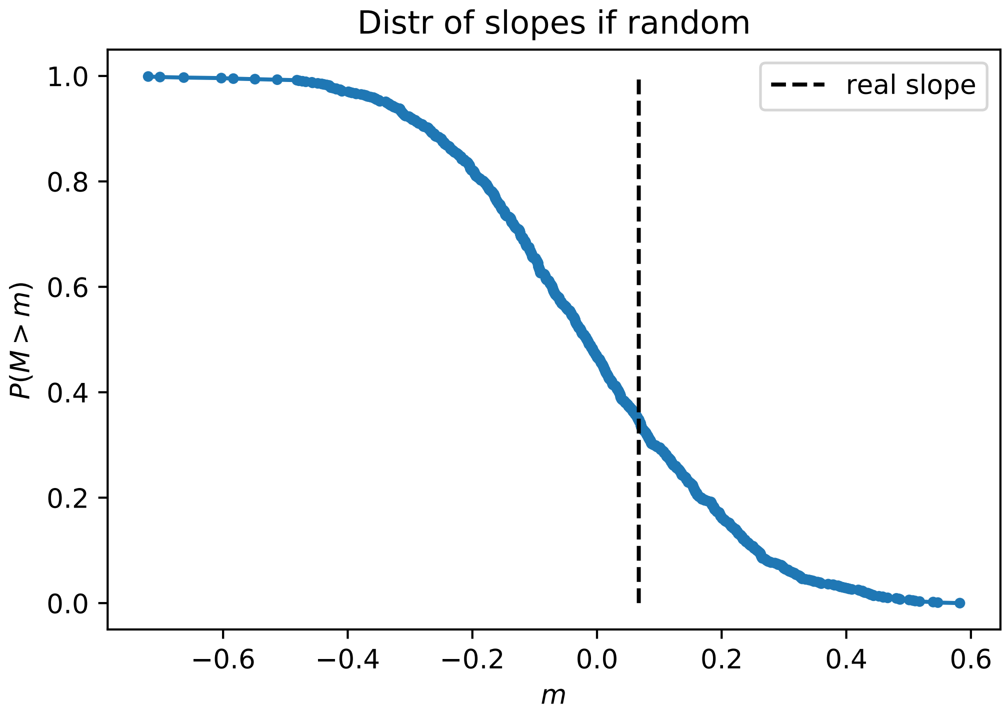
Looks *a bit* different... we should do some quantitative statistics and not rely only on visuals...

[6]:
```
# data:
x = 2 * np.random.randn(100) + 0.5
y = 4*np.random.randn(100) + 0.5*x


# linear regression x and y
slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x,y)


# randomize Y vs X many times, get slope of each:
list_r_slopes = []
for _ in range(1000):
    yr = y.copy()
    np.random.shuffle(yr)

    slope_r = scipy.stats.linregress(x,yr)[0]
    list_r_slopes.append(slope_r)
```

And plot:

[7]:
```
# plot cdf of slopes:
x_cdf = sorted(list_r_slopes)
N = len(x_cdf)
y_cdf = [ (N-1.0-i)/N for i in range(N) ]

plt.title("Distr of slopes if random")
```

```
plt.plot(x_cdf, y_cdf, '.-')
plt.plot( [slope, slope], [0,1], 'k--', label='real slope' ) # dashed line for true␣
↪slope
plt.xlabel("$m$");plt.ylabel("$P(M>m)$"); plt.legend();
```



And we can use these simulations to estimate the **probability that we see a slope at least as big as the real slope in our random data** (under our null hypothesis):

- This is the dreaded p-value!
- Use "hat" notation, $\hat{p}$ for empirical estimate.

[8]:
```
p_hat = 1.0*len([si for si in list_r_slopes if si >= slope]) / len(list_r_slopes)

print("    p =", p_value)
print("p_hat =", p_hat)
```

```
    p = 0.747193462469739
p_hat = 0.349
```

Oops, $\hat{p}$ looks to be something like **half** the size of p. What happened?

- Two-tailed tests vs. one-tail. We should ask what is the probability of getting a slope **as extreme** as the one observed:

$$\Pr(|M| > m) = \Pr(M < -m) + \Pr(M > m)$$

[9]:
```
p_hat2 = 1.00*len([si for si in list_r_slopes if abs(si) >= slope]) / len(list_r_slopes)
```

```python
print("     p =", p_value)
print("p_hat2 =", p_hat2)
```

```
     p = 0.747193462469739
p_hat2 = 0.762
```

Not bad!

- Linear regression is another example where the analytic distribution of $P(|M| > m)$ under the null hypothesis is known, so we can use the special functions. (Usually the slope is converted into a t-statistic by normalizing by the standard error $\rightarrow$ t-test.)

BTW, the notion of **permutation testing** is well established. This name usually means one needs to try **every** permutation of the data, and it is also called an **exact test**. It is as strict as the data allow, but if you have too many points, the number of permutation is far too big to enumerate all of them. Hence we randomly sample the permutation space and call what we are doing **monte carlo permutation testing**.

Linear regression

First, let's **recap**:

The goal with linear regression was to find the line that best fit some given XY-data (and test if that fit was significant). For the ith data point:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

We want to find the pair of values (β_0,β_1) that minimize the sum of square errors S:

$$S(\beta) = \sum_i \epsilon_i^2 = \sum_i \left[y_i - (\beta_0 + \beta_1 x_i)\right]^2$$

Of course, this holds for all the data points together:

$$y_1 = \beta_0 + \beta_1 x_1 + \epsilon_1 y_2 = \beta_0 + \beta_1 x_2 + \epsilon_2 y_3 = \beta_0 + \beta_1 x_3 + \epsilon_3 \vdots y_n = \beta_0 + \beta_1 x_n + \epsilon_n$$

We can write this in matrix notation:

Now a single matrix equation captures all individual linear regression equations. Furthermore, if we want to estimate the β's with Ordinary Least Squares (OLS), we have a (relatively) simple way to write down the best β's.

In matrix form the residuals are:

$$\epsilon = \mathbf{y} - \mathbf{X}\beta$$

The sum of the squared residuals is then (using the inner product):

$$\epsilon^{\mathrm{T}}\epsilon = (\mathbf{y} - \mathbf{X}\beta)^{\mathrm{T}} (\mathbf{y} - \mathbf{X}\beta)$$

where ϵ^{T} is the transpose of ϵ.

With a little calculus and elbow grease we can find the minimum of this equation, which can be solved to give us the best estimates for β, called $\hat{\beta}$:

$$\hat{\beta} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

assuming $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is nonsingular, meaning we can compute the *matrix inverse*.

We already computed the values of β_0 and β_1, so why are we now getting into all this **matrix nonsense**?

- Because the above derivation is not limited to two coefficients and a linear regression of the form $y = \beta_0 + \beta_1 x$.
- Until now, we've been doing **simple linear regression**.

Multiple linear regression.

There is no reason why we need to stop our equation at two coefficients. What about this:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

In matrix form that is exactly the same equation:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

except that $\mathbf{X}$ has $p+1$ columns and β has $p+1$ rows.

- This means we can solve exactly the same linear regression but we are not limited to a single column of x data!

For example, we may have reason to study a linear model relating a person's height and weight:

$$\text{weight} = \beta_0 + \beta_1 \text{height}$$

Now we can also incorporate (probably unrealistically) age data as well:

$$\text{weight} = \beta_0 + \beta_1 \text{height} + \beta_2 \text{age}$$

We have a new coefficient, and we have a new column (the age column) in our data matrix. But estimating $\hat{\beta}$ remains the same.

A note on jargon

There's a **lot** of names for the quantities in these equations, it's hard to keep straight.

The y_i's are called:

- Regressand
- endogenous variable

- response variable
- dependent variable

The x_i's (including the constant 1) are called:

- Regressors
- exogenous variables
- explanatory variables
- independent variables

The matrix of data **X** is called the **design matrix**.

The β's are called:

- effects
- regression coefficients
- regression parameters

The vector **fi** is called the **parameter** or **coefficient vector**.

The ϵ's are called **errors**, **error term**, or **noise**.

Handy mnemonic for the x's and y's: The x's are the exogenous variable because "exogenous" contains "x".

Let's see this in action:

We're going to use a new library called statsmodels.

First let's generate some data:

[10]:
```
nobs = 100
b = [1.0,0.1,0.5] # betas
Xlist = []
Ylist = []
for _ in range(nobs):
    # build the individual observation:
    x1 = np.random.random() # a single number
    x2 = np.random.random()
    e = np.random.randn()*0.05 # random noise
    y = 1.0*b[0] + x1*b[1] + x2*b[2] + e

    # add it to the matrices:
    Xlist.append([1.0,x1,x2]) # design matrix/exog variables
    Ylist.append(y) # endogenous variables
```

Let's even print the first few lines:

[11]:
```
for i in range(5):
    print(Ylist[i], "-->", Xlist[i])
print()
print("Size Y =", len(Ylist))
print("Size X =", len(Xlist))
```

```
1.0973425374984798 --> [1.0, 0.45018414737870416, 0.11155703396591798]
1.2489886615452195 --> [1.0, 0.3370883422037583, 0.505925226932003]
1.2585253352599732 --> [1.0, 0.9657832552602018, 0.4073882862479439]
1.5021723883264246 --> [1.0, 0.885917604994843, 0.8058611034443959]
```

```
1.0578151920207313 --> [1.0, 0.5224070055934384, 0.18337687427663074]

Size Y = 100
Size X = 100
```

Looks good.

If you're comfortable with matrix operations, you can also build the data using matrix multiplication:

[12]:
```
nobs= 25
X = np.hstack(( np.ones((nobs,1)), np.random.random((nobs,2)) ))

beta = [1, 0.1, 0.5]
e = np.random.randn(nobs)*0.05
y = np.dot(X, beta) + e

print("(rows, columns)")
print(X.shape)
print(y.shape)
```

```
(rows, columns)
(25, 3)
(25,)
```

But matrix operations are not very comfortable in Python (at least compared to MATLAB), so you might just want to avoid them. It's up to you.

Anyway, now that we've got the data generated, let's use statsmodels `OLS` function to find the **f̂i**:

[13]:
```
import statsmodels.api as sm

# Fit regression model
model = sm.OLS(y,X) # our matrix data
#model = sm.OLS(Ylist, Xlist) # our lists of data

result = model.fit()
```

[14]:
```
print(result.summary2()) # summary2() narrow than summary()
```

```
                 Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.926
Dependent Variable: y                AIC:                -81.9373
Date:               2019-10-31 10:25 BIC:                -78.2806
No. Observations:   25               Log-Likelihood:     43.969
Df Model:           2                F-statistic:        152.2
Df Residuals:       22               Prob (F-statistic): 1.31e-13
R-squared:          0.933            Scale:              0.0019743
---------------------------------------------------------------------
          Coef.     Std.Err.       t       P>|t|     [0.025    0.975]
---------------------------------------------------------------------
const     0.9940      0.0197    50.5770    0.0000    0.9532    1.0347
x1        0.1018      0.0317     3.2089    0.0040    0.0360    0.1676
x2        0.5299      0.0355    14.9452    0.0000    0.4563    0.6034
```

```
==============================================================================
Omnibus:                        3.328   Durbin-Watson:                   2.040
Prob(Omnibus):                  0.189   Jarque-Bera (JB):                1.982
Skew:                           0.667   Prob(JB):                        0.371
Kurtosis:                       3.348   Condition No.:                       5
==============================================================================
```

This type of display is very common across statistical packages. There's three main panels.

- The top panel contains information about the size of the data and the overall accuracy of the fit, as well as some nice records like time of the calculation.
- The middle panel contains information about each **coefficient** of the fit.
- The bottom panel contains information about the residuals ϵ_i of the fit, are they normally distributed (omnibus test), do they possess serial correlations (Durbin-Watson), do the residuals have the same skewness/kurtosis as a normal distribution (Jarque-Bera), etc.
  - Condition Number - When it's big (like in the 1000s), this warns us there's something wrong with our design matrix, often multicollinearity.

We can also access the information shown in the summary directly:

[15]:
```
print(result.params) # the coefficients
print(b, "(true)")
print(result.rsquared)
```

```
[0.99395605 0.10179811 0.52986259]
[1.0, 0.1, 0.5] (true)
0.9325796733091033
```

Here are all the things `result` has:

[16]:
```
for name in dir(result):
    if "__" in name: # skip "private" stuff
        continue
    print("  ", name)
```

```
   HC0_se
   HC1_se
   HC2_se
   HC3_se
   _HCCM
   _cache
   _data_attr
   _get_robustcov_results
   _is_nested
   _wexog_singular_values
   aic
   bic
   bse
   centered_tss
   compare_f_test
   compare_lm_test
   compare_lr_test
```

```
condition_number
conf_int
conf_int_el
cov_HC0
cov_HC1
cov_HC2
cov_HC3
cov_kwds
cov_params
cov_type
df_model
df_resid
eigenvals
el_test
ess
f_pvalue
f_test
fittedvalues
fvalue
get_influence
get_prediction
get_robustcov_results
initialize
k_constant
llf
load
model
mse_model
mse_resid
mse_total
nobs
normalized_cov_params
outlier_test
params
predict
pvalues
remove_data
resid
resid_pearson
rsquared
rsquared_adj
save
scale
ssr
summary
summary2
t_test
t_test_pairwise
tvalues
uncentered_tss
use_t
wald_test
```

```
wald_test_terms
wresid
```

There's lots of data we can interact with regarding the fit...

ASIDE

Here's an example of a bad design matrix:

[17]:
```python
nobs = 100
b = [1.0,0.1,0.5] # betas
Xlist = []
Ylist = []
for _ in range(nobs):
    # build the individual observation:
    x1 = np.random.random() # a single number
    x2 = 12.7 * x1 + 0.1*np.random.random()  # DUN DUN DUNNNNNNN!
    e = np.random.randn()*0.05 # random noise
    y = 1.0*b[0] + x1*b[1] + x2*b[2] + e

    # add it to the matrices:
    Xlist.append([1.0,x1,x2]) # design matrix/exog variables
    Ylist.append(y) # endogenous variables
```

And fit:

[18]:
```python
model = sm.OLS(Ylist, Xlist)
result = model.fit()
print(result.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.999
Model:                            OLS   Adj. R-squared:                  0.999
Method:                 Least Squares   F-statistic:                 7.088e+04
Date:                Thu, 31 Oct 2019   Prob (F-statistic):          3.12e-154
Time:                        10:25:51   Log-Likelihood:                 166.87
No. Observations:                 100   AIC:                            -327.7
Df Residuals:                      97   BIC:                            -319.9
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.9905      0.013     76.422      0.000       0.965       1.016
x1            -0.2577      2.104     -0.122      0.903      -4.433       3.917
x2             0.5287      0.166      3.191      0.002       0.200       0.858
==============================================================================
Omnibus:                        1.444   Durbin-Watson:                   1.705
Prob(Omnibus):                  0.486   Jarque-Bera (JB):                1.168
Skew:                          -0.264   Prob(JB):                        0.558
Kurtosis:                       3.038   Cond. No.                     3.31e+03
==============================================================================
```

```
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 3.31e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The design matrix has a problem. Two of the exogenous variables are collinear: $x_2 \approx 12.7x_1$.

- This indicates bad experimental design (you've accidentally measured the same thing twice, essentially)
- This prevents a unique solution for the least-squares linear regression. You can't compute the inverse when solving

$$\hat{\beta} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1} \mathbf{X}^{\mathrm{T}}\mathbf{y}$$

- More precisely, the matrix is **ill-conditioned** because, while I did get an answer for $\hat{\beta}$, a tiny change somewhere in $\mathbf{X}$ will completely change the value of $\hat{\beta}$

Let's see how "stable" the solution is by running a fit on another batch of data and looking at β_1 and β_2 again:

[19]:

```python
Xlist = []
Ylist = []
for _ in range(nobs):
    # build the individual observation:
    x1 = np.random.random() # a single number
    x2 = 12.7 * x1 + 0.1*np.random.random()  # DUN DUN DUNNNNNNN!
    e = np.random.randn()*0.05 # random noise
    y = 1.0*b[0] + x1*b[1] + x2*b[2] + e

    # add it to the matrices:
    Xlist.append([1.0,x1,x2]) # design matrix/exog variables
    Ylist.append(y) # endogenous variables

model = sm.OLS(Ylist, Xlist)
result2 = model.fit()
print(result2.summary() )
print()
print(" result betas =", result.params)
print("result2 betas =", result2.params)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.999
Model:                            OLS   Adj. R-squared:                  0.999
Method:                 Least Squares   F-statistic:                 5.849e+04
Date:                Thu, 31 Oct 2019   Prob (F-statistic):          3.44e-150
Time:                        10:25:51   Log-Likelihood:                 153.19
No. Observations:                 100   AIC:                            -300.4
Df Residuals:                      97   BIC:                            -292.6
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
```

```
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.9926      0.016     60.925      0.000       0.960       1.025
x1             0.0562      2.517      0.022      0.982      -4.939       5.051
x2             0.5034      0.198      2.539      0.013       0.110       0.897
==============================================================================
Omnibus:                        4.957   Durbin-Watson:                   1.903
Prob(Omnibus):                  0.084   Jarque-Bera (JB):                2.704
Skew:                           0.153   Prob(JB):                        0.259
Kurtosis:                       2.255   Cond. No.                     3.53e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 3.53e+03. This might indicate that there are
strong multicollinearity or other numerical problems.

 result betas = [ 0.99052339 -0.25768364  0.52868301]
result2 betas = [0.99259273 0.05624711 0.50343679]
```

[20]:
```python
const, x1_list, x2_list = zip(*Xlist)

print(scipy.stats.pearsonr(x1_list,x2_list))
```

```
(0.9999719510839012, 4.0113495735965905e-210)
```

Here's another example. In this case we are going to use `sm.add_constant` to more quickly add the column of ones to the design matrix.

- This uses real data as well so we don't need to add fake ϵ's ourselves.
- But let's add a **random variable** and see if the model rejects it…

[21]:
```python
Height    = [1.47,1.50,1.52,1.55,1.57,1.60,1.63,1.65,
             1.68,1.70,1.73,1.75,1.78,1.80,1.83]

Unrelated = list(np.random.random(len(Height))-1)


Weight    = [52.21,53.12,54.48,55.84,57.20,58.57,59.93,
             61.29,63.11,64.47,66.28,68.10,69.92,72.19,74.46]

# stack exog variables into design matrix
X = np.column_stack( (Height,Unrelated) )
X = sm.add_constant(X, prepend=True) # add a column of 1's out front


res = sm.OLS(Weight,X).fit() # create a model and fit it

print(res.summary())
```

```
                            OLS Regression Results
==============================================================================
```

```
Dep. Variable:                      y   R-squared:                       0.990
Model:                            OLS   Adj. R-squared:                  0.988
Method:                 Least Squares   F-statistic:                     566.2
Date:                Thu, 31 Oct 2019   Prob (F-statistic):           1.33e-12
Time:                        10:25:51   Log-Likelihood:                -15.853
No. Observations:                  15   AIC:                             37.71
Df Residuals:                      12   BIC:                             39.83
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        -39.0701      3.013    -12.968      0.000     -45.635     -32.506
x1            61.1121      1.840     33.204      0.000      57.102      65.122
x2            -0.4837      0.804     -0.602      0.559      -2.235       1.268
==============================================================================
Omnibus:                        1.683   Durbin-Watson:                   0.485
Prob(Omnibus):                  0.431   Jarque-Bera (JB):                1.242
Skew:                           0.507   Prob(JB):                        0.537
Kurtosis:                       2.021   Cond. No.                         35.3
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

```
/anaconda3/lib/python3.7/site-packages/scipy/stats/stats.py:1416: UserWarning:
kurtosistest only valid for n>=20 ... continuing anyway, n=15
  "anyway, n=%i" % int(n))
```

Let's make a quickie plot of the solution and the data:

[22]:
```
from mpl_toolkits.mplot3d import Axes3D
from JSAnimation import IPython_display
from matplotlib import animation

fig = plt.figure()
ax  = Axes3D(fig)

# build the linear fit (plane):
x1 = np.linspace(min(Height),  max(Height),  5)
x2 = np.linspace(min(Unrelated),max(Unrelated),5)
x1, x2 = np.meshgrid(x1,x2)
b0,b1,b2 = res.params
y = b0 + b1*x1 + b2*x2

print("The plane is defined by y = %f + %f x1 + %f x2" % (b0,b1,b2))

ax.scatter(Height,Unrelated,Weight, c='r', s=50)
ax.plot_surface(x1,x2,y, alpha=0.5, lw=0)

ax.set_xlabel("Height",   labelpad=20)
ax.set_ylabel("Unrelated", labelpad=20)
```

```python
ax.set_zlabel("Weight",   labelpad=20)

ax.tick_params(axis='both',which='major', labelsize=14)

# point-of-view: (altitude degrees, azimuth degrees)
ax.view_init(30, 0)

def animate(t):
    # update POV:
    ax.view_init(30 + np.sin(2*np.pi*t/120.0)*30,
                 3*t )
    return []

animation.FuncAnimation(fig, animate,
                        frames=120,  # number of frames to draw
                        interval=40, # time (ms) on each frame
                        blit=True)
```

```
The plane is defined by y = -39.070060 + 61.112146 x1 + -0.483739 x2
```

[22]:
```
<matplotlib.animation.FuncAnimation at 0x7ff898e8fa90>
```

OK, so we see that, since the “Unrelated” axis isn’t meaningful, our regression is essentially a flat plane. Let’s do an example with real data, where both variables matter:

[23]:
```python
data = np.genfromtxt("anaesthesia_1959.csv", dtype=float,
                     delimiter=',', names=True)

# load each column into separate array:
LogDose   = data["LogDose"]
BP        = data["BP"]
RecovTime = data["RecovTime"]
```

This data measures the time to recovery from anaesthetic as a function of blood pressure and (the logarithm) of the dosage.

Fit it up!!!

[24]:
```python
# stack exog variables into design matrix
X = np.column_stack( (LogDose,BP/10.0) )
X = sm.add_constant(X, prepend=True) # add a column of 1's out front



res = sm.OLS(RecovTime,X).fit() # create a model and fit it

print(res.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.228
Model:                            OLS   Adj. R-squared:                  0.197
Method:                 Least Squares   F-statistic:                     7.364
Date:                Thu, 31 Oct 2019   Prob (F-statistic):            0.00157
```

```
Time:                        10:25:58   Log-Likelihood:                -214.45
No. Observations:                  53   AIC:                             434.9
Df Residuals:                      50   BIC:                             440.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         22.2712     17.555      1.269      0.210     -12.989      57.531
x1            10.6399      2.856      3.726      0.000       4.904      16.376
x2            -7.4009      2.893     -2.558      0.014     -13.212      -1.590
==============================================================================
Omnibus:                        7.999   Durbin-Watson:                   1.729
Prob(Omnibus):                  0.018   Jarque-Bera (JB):                7.263
Skew:                           0.870   Prob(JB):                       0.0265
Kurtosis:                       3.510   Cond. No.                         74.0
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Now, let's plot:

[25]:
```
fig = plt.figure(figsize=(8,5))
ax  = Axes3D(fig)

x = np.linspace(min(LogDose),max(LogDose),50)
y = np.linspace(min(BP/10.0),max(BP/10.0),50)
x, y = np.meshgrid(x,y)
z = 22.2712 + 10.6399*x -7.4009*y

ax.scatter(LogDose,BP/10.0,RecovTime, c='r', s=50)
ax.plot_surface(x,y,z, alpha=0.5, lw=0)

ax.set_xlabel("LogDose", labelpad=20)
ax.set_ylabel("Blood Pressure/10", labelpad=20)
ax.set_zlabel("Recovery Time", labelpad=20)
ax.tick_params(axis='both',which='major', labelsize=14)



# point-of-view: (altitude degrees, azimuth degrees)
ax.view_init(30, 0)

def animate(t):
    # update POV:
    ax.view_init(30 + np.sin(2*np.pi*t/120.0)*40, 3*t )
    return []

animation.FuncAnimation(fig, animate,
                        frames=120,  # number of frames to draw
```

```
                              interval=40, # time (ms) on each frame
                              blit=True)
```

[25]: <matplotlib.animation.FuncAnimation at 0x7ff87857f3c8>

In this case, since both BP and LogDose are significant, the plane of the best fit function is tilted in both the X and Y directions.

Named variables and generality of linear regression

There's one issue with the previous summaries, which is that we need to **remember** that `x1` is `Height` and `x2` is `Unrelated`, or whatever.

`Statsmodels` provides another very compact way of specifying a linear fit, but to use it we need to play around with another data structure (introduced by R) called a `DataFrame`. Python dataframes are provided by a nice third-party package called pandas

Let's convert the Height/Weight/Unrelated data to a Pandas dataframe and then we'll look at it to see what it does:

[26]:
```
import pandas as pd

data = list(zip(Height,Unrelated,Weight))
df = pd.DataFrame(data,
                  columns=["Height","Unrelated","Weight"])

print(df)
```

```
    Height  Unrelated  Weight
0     1.47  -0.682098   52.21
1     1.50  -0.525703   53.12
2     1.52  -0.001378   54.48
3     1.55  -0.712611   55.84
4     1.57  -0.717378   57.20
5     1.60  -0.628620   58.57
6     1.63  -0.830683   59.93
7     1.65  -0.261008   61.29
8     1.68  -0.620660   63.11
9     1.70  -0.419141   64.47
10    1.73  -0.338697   66.28
11    1.75  -0.964556   68.10
12    1.78  -0.355997   69.92
13    1.80  -0.467231   72.19
14    1.83  -0.917150   74.46
```

The Pandas `DataFrame` and `Series` objects provide very nice ways to deal with missing data, perform row/column-selects, and more.

Pandas describes the `DataFrame` as

> [A] Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels.

The named columns are what we want. `Statsmodels` can automagically use them to build regression models.

- Note: this uses `statsmodels.formula.api` not `statsmodels.api`

[27]:
```
import statsmodels.formula.api as smf


mod = smf.ols(formula='Weight ~ Height + Unrelated',
              data=df)

res = mod.fit()
print(res.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 Weight   R-squared:                       0.990
Model:                            OLS   Adj. R-squared:                  0.988
Method:                 Least Squares   F-statistic:                     566.2
Date:                Thu, 31 Oct 2019   Prob (F-statistic):           1.33e-12
Time:                        10:26:47   Log-Likelihood:                -15.853
No. Observations:                  15   AIC:                             37.71
Df Residuals:                      12   BIC:                             39.83
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -39.0701      3.013    -12.968      0.000     -45.635     -32.506
Height        61.1121      1.840     33.204      0.000      57.102      65.122
Unrelated     -0.4837      0.804     -0.602      0.559      -2.235       1.268
==============================================================================
Omnibus:                        1.683   Durbin-Watson:                   0.485
Prob(Omnibus):                  0.431   Jarque-Bera (JB):                1.242
Skew:                           0.507   Prob(JB):                        0.537
Kurtosis:                       2.021   Cond. No.                         35.3
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

```
/anaconda3/lib/python3.7/site-packages/scipy/stats/stats.py:1416: UserWarning:
kurtosistest only valid for n>=20 ... continuing anyway, n=15
  "anyway, n=%i" % int(n))
```

Everything about this is pretty much the same, except we very quickly specified the linear model using a string!

- `"Weight ~ Height + Unrelated"`

The formula works very much like an equation, but the tilde ("~") separates the right and left sides of the equation and denotes that there is a constant term as well (think the two sides are "proportional"). This term is called "Intercept" in the previous summary.

The power of the formula description is that it lets us go well **beyond** linear regression:

[28]:
```
url = "http://vincentarelbundock.github.com/Rdatasets/csv/HistData/Guerry.csv"
df = pd.read_csv(url)
df = df[['Lottery', 'Literacy', 'Wealth', 'Region']].dropna()
print(df.head())
```

```
   Lottery  Literacy  Wealth Region
0       41        37      73      E
1       38        51      22      N
2       66        13      61      C
3       80        46      76      E
4       79        69      83      E
```

Multiplicative effects:

[29]:
```
res = smf.ols(formula='Lottery ~ Literacy * Wealth - 1', data=df).fit()
print(res.summary2())
```

```
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.811
Dependent Variable: Lottery          AIC:                766.2917
Date:               2019-10-31 10:26 BIC:                773.6197
No. Observations:   85               Log-Likelihood:     -380.15
Df Model:           3                F-statistic:        122.3
Df Residuals:       82               Prob (F-statistic): 3.55e-30
R-squared:          0.817            Scale:              465.29
-----------------------------------------------------------------
                   Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
-----------------------------------------------------------------
Literacy           0.4274   0.0995  4.2974 0.0000  0.2295  0.6252
Wealth             1.0810   0.1040 10.3970 0.0000  0.8742  1.2878
Literacy:Wealth   -0.0136   0.0032 -4.2647 0.0001 -0.0200 -0.0073
-----------------------------------------------------------------
Omnibus:               2.001        Durbin-Watson:           1.946
Prob(Omnibus):         0.368        Jarque-Bera (JB):        2.002
Skew:                  -0.321       Prob(JB):                0.367
Kurtosis:              2.609        Condition No.:           90
=================================================================
```

Nonlinear functions:

[30]:
```
res = smf.ols(formula='Lottery ~ np.log(Literacy)', data=df).fit()
print(res.summary2())
```

```
                  Results: Ordinary least squares
===================================================================
Model:                OLS              Adj. R-squared:     0.151
Dependent Variable:   Lottery          AIC:                774.7658
Date:                 2019-10-31 10:26 BIC:                779.6511
No. Observations:     85               Log-Likelihood:     -385.38
Df Model:             1                F-statistic:        15.89
```

```
Df Residuals:         83              Prob (F-statistic): 0.000144
R-squared:            0.161           Scale:              519.97
-------------------------------------------------------------------
                  Coef.   Std.Err.    t    P>|t|   [0.025   0.975]
-------------------------------------------------------------------
Intercept        115.6091  18.3742  6.2919 0.0000  79.0637 152.1546
np.log(Literacy) -20.3940   5.1163 -3.9861 0.0001 -30.5701 -10.2178
-------------------------------------------------------------------
Omnibus:                8.907       Durbin-Watson:           2.019
Prob(Omnibus):          0.012       Jarque-Bera (JB):        3.299
Skew:                   0.108       Prob(JB):                0.192
Kurtosis:               2.059       Condition No.:           29
===================================================================
```

Summary

If we have reason to believe a linear model is sufficient to explain a relationship between numeric data, we don't need to be restricted to 1D functions.

The "summary" view shown here is standardized across many different software packages, including those more focused on statistical analysis than Python (R is the main example).

- Learning to read these summaries makes it useful to see which coefficients are/are not significant.