Refreshing Your Linear Algebra Knowledge with NumPy, Part II

Steve Lindblad

DataSciPy Meetup
Veritas Technologies LLC
Roseville, MN
November 10, 2018

[SL (11/10/2018): This is the notebook as produced live from my talk today. Note: "Part I" was my presentation of similar material at the 7/12/2018 PyMNtos meeting. It consisted essentially of the same material below but ended just before the "LU Decomposition" section.]

In [ ]:

```

```

Example

System of 3 linear equations in 3 unknowns (variables):

$$\begin{aligned} 2x + 2y + 4z &= 0 \\ y - 5z &= 13 \\ 3y + 4z &= 1 \end{aligned}$$

Goal is to find the values of x, y, and z that make all three equations true simultaneously.

One way to solve is to manipulate the equations directly:

(1) Subtract 3 times the second equation from the third to eliminate y and solve for z:

$$\begin{array}{rcrcr} 3y & + & 4z & = & 1 \\ -3y & + & 15z & = & -39 \\ \hline & & 19z & = & -38 \\ & & z & = & -2 \end{array}$$

(2) Plug $z = -2$ back into the second equation to solve for y:

$$\begin{aligned} y - 5(-2) &= 13 \\ y + 10 &= 13 \\ y &= 3 \end{aligned}$$

(3) Plug $y = 3$ and $z = -2$ into the first equation to solve for x:

$$\begin{aligned} 2x + 2(3) + 4(-2) &= 0 \\ 2x &= 2 \\ x &= 1 \end{aligned}$$

Disadvantage

Above approach is much more difficult with larger systems of equations:

$$\begin{aligned} 2v + 7w - x + 3y + 6z &= 24 \\ 3v + 3w + x - y + 5z &= -1 \\ v - w - 2x - 3y - 8z &= -16 \\ 4v + 5w - 3x + 7y + 10z &= 52 \\ -5v + 3w - 9x - 9y + z &= 25 \end{aligned}$$

Power of matrix notation

$$\begin{array}{ccccc} \begin{bmatrix} 2 & 7 & -1 & 3 & 6 \\ 3 & 3 & 1 & -1 & 5 \\ 1 & -1 & -2 & -3 & -8 \\ 4 & 5 & -3 & 7 & 10 \\ -5 & 3 & -9 & -9 & 1 \end{bmatrix} & \times & \begin{bmatrix} v \\ w \\ x \\ y \\ z \end{bmatrix} & = & \begin{bmatrix} 24 \\ -1 \\ -16 \\ 52 \\ 25 \end{bmatrix} \\ A & \times & t & = & d \end{array}$$

Using Matrix Techniques to Solve a Linear System

First, let's first return to the original system:

$$\begin{aligned} 2x + 2y + 4z &= 0 \\ y - 5z &= 13 \\ 3y + 4z &= 1 \end{aligned}$$

or

$$\underbrace{\begin{bmatrix} 2 & 2 & 4 \\ 0 & 1 & -5 \\ 0 & 3 & 4 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}_{t} = \underbrace{\begin{bmatrix} 0 \\ 13 \\ 1 \end{bmatrix}}_{d}$$

Step 0

Form the **augmented matrix**:

$$\left[\begin{array}{ccc:c} 2 & 2 & 4 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{array}\right]$$

Enter data into Python

In [2]:
```
import numpy as np
```

In [3]:
```
A = np.matrix("2 2 4; 0 1 -5; 0 3 4")
```

In [4]:
```
d = np.mat("0; 13; 1")
```

In [5]:
```
A
```

Out[5]:
```
matrix([[ 2,  2,  4],
        [ 0,  1, -5],
        [ 0,  3,  4]])
```

In [6]:
```
d
```

Out[6]:
```
matrix([[ 0],
        [13],
        [ 1]])
```

In [7]:
```
M = np.hstack( (A,d) )
```

In [8]:
```
M
```

Out[8]:
```
matrix([[ 2,  2,  4,  0],
        [ 0,  1, -5, 13],
        [ 0,  3,  4,  1]])
```

Goal

Given the augmented matrix

$$\left[\begin{array}{ccc:c} 2 & 2 & 4 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{array}\right]$$

we want to perform **elementary row operations** (Gaussian elimination) to transform the above into an equivalent matrix of the the following form, from which the solution can be read:

$$\left[\begin{array}{ccc:c} 1 & 0 & 0 & ? \\ 0 & 1 & 0 & ? \\ 0 & 0 & 1 & ? \end{array}\right]$$

or

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix}$$

or

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix}$$

Elementary row operations (Gaussian elimination)

- **multiply** a row by a non-zero scalar
- **add to** one row a scalar multiple of another row
- **interchange** of two rows

Step 1

First step is multiply the first row by $\frac{1}{2}$ to get a 1 in the first column:

$$\begin{bmatrix} 2 & 2 & 4 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 1 & 2 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{bmatrix}$$

Step 1 as a matrix multiplication

The idea of "multiply the first row by ½" can be expressed as a matrix multiplication:

$$\begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2 & 4 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{bmatrix}$$

Matrix on the left, an identity matrix with a small adjustment, is a **elementary matrix**.

Create a Python function to return the elementary matrix:

In [9]:
```
def scalerow(r, α, n=3):
    """Elementary matrix to multiply row r by the scalar α,
when multiplied on the left of a target matrix of n rows."""
    E = np.asmatrix(np.eye(n))
    E[r,r] = α
    return E
```

In [12]:
```
E1 = scalerow(0, .5); E1
```

Out[12]:
```
matrix([[0.5, 0. , 0. ],
        [0. , 1. , 0. ],
        [0. , 0. , 1. ]])
```

In [13]:
```
E1*M
```

Out[13]:
```
matrix([[ 1.,  1.,  2.,  0.],
        [ 0.,  1., -5., 13.],
        [ 0.,  3.,  4.,  1.]])
```

In [ ]:

[SL (11/10/2018): The following was in response to a question during the talk. This confirms that np.matrix() can be used in place of np.asmatrix() to convert a data array to a matrix.]

In [11]:
```
np.matrix(np.eye(3))
```

Out[11]:
```
matrix([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

In [ ]:

In [ ]:

Step 2

Next, subtract 3 times row 1 from row 2.

$$\begin{bmatrix} 1 & ? & ? \\ ? & 1 & ? \\ ? & ? & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 2 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 0 & 19 & -38 \end{bmatrix}$$

In [14]:
```
def addtorow(r, α, j, n=3):
    """Elementary matrix to add α times row j to row r,
when multiplied on the left of a target matrix of n rows."""
    E = np.asmatrix(np.eye(n))
    E[r,j] = α
    return E
```

In [15]:
```
E2 = addtorow(2, -3, 1); E2
```

Out[15]:
```
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0., -3.,  1.]])
```

In [16]:
```
E2*E1*M
```

Out[16]:
```
matrix([[  1.,   1.,   2.,   0.],
        [  0.,   1.,  -5.,  13.],
        [  0.,   0.,  19., -38.]])
```

Remaining row operation steps

In [17]:
```
E3 = addtorow(0, -1, 1); E3*E2*E1*M
```

Out[17]:
```
matrix([[  1.,   0.,   7., -13.],
        [  0.,   1.,  -5.,  13.],
        [  0.,   0.,  19., -38.]])
```

In [18]:
```
E4 = scalerow(2, 1/19); E4*E3*E2*E1*M
```

Out[18]:
```
matrix([[  1.,   0.,   7., -13.],
        [  0.,   1.,  -5.,  13.],
        [  0.,   0.,   1.,  -2.]])
```

In [19]:
```
E5 = addtorow(1, 5, 2); E5*E4*E3*E2*E1*M
```

Out[19]:
```
matrix([[ 1.00000000e+00,  0.00000000e+00,  7.00000000e+00,
         -1.30000000e+01],
        [ 0.00000000e+00,  1.00000000e+00, -2.22044605e-16,
          3.00000000e+00],
        [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00,
         -2.00000000e+00]])
```

In [20]:
```
E6 = addtorow(0, -7, 2); E6*E5*E4*E3*E2*E1*M
```

Out[20]:
```
matrix([[ 1.00000000e+00, -5.55111512e-17,  4.44089210e-16,
          1.00000000e+00],
        [ 0.00000000e+00,  1.00000000e+00, -2.22044605e-16,
          3.00000000e+00],
        [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00,
         -2.00000000e+00]])
```

In [21]:
```
np.round( E6*E5*E4*E3*E2*E1*M  ,9)
```

Out[21]:
```
array([[ 1., -0.,  0.,  1.],
       [ 0.,  1., -0.,  3.],
       [ 0.,  0.,  1., -2.]])
```

Summary

We have transformed the original augmented matrix

$$\left[\begin{array}{rrr:r} 2 & 2 & 4 & 0 \\ 0 & 1 & -5 & 13 \\ 0 & 3 & 4 & 1 \end{array}\right]$$

to the equivalent augmented matrix

$$\left[\begin{array}{rrr:r} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -2 \end{array}\right].$$

In other words,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix}$$

Or

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix}$$

Thus, $x = 1$, $y = 3$, and $z = -2$ is the solution to the original system of equations.

The point $(1, 3, -2)$ in xyz-space is the intersection of the planes given by the three equations.

In [ ]:
```

```

Revisiting the elementary matrices

In [22]:
```
np.round( (E6*E5*E4*E3*E2*E1) * A ,9)
```

Out[22]:
```
array([[ 1., -0.,  0.],
       [ 0.,  1., -0.],
       [ 0.,  0.,  1.]])
```

In [23]:
```
A
```

Out[23]:
```
matrix([[ 2,  2,  4],
        [ 0,  1, -5],
        [ 0,  3,  4]])
```

In [24]:
```
E6*E5*E4*E3*E2*E1
```

Out[24]:
```
matrix([[ 0.5       ,  0.10526316, -0.36842105],
        [ 0.        ,  0.21052632,  0.26315789],
        [ 0.        , -0.15789474,  0.05263158]])
```

In [25]:
```
A.I
```

Out[25]:
```
matrix([[ 0.5       ,  0.10526316, -0.36842105],
        [ 0.        ,  0.21052632,  0.26315789],
        [-0.        , -0.15789474,  0.05263158]])
```

Answer (version 2)

In [26]:
```
(E6*E5*E4*E3*E2*E1) * d
```

Out[26]:
```
matrix([[ 1.],
        [ 3.],
        [-2.]])
```

In [27]:
```
A.I*d
```

Out[27]:
```
matrix([[ 1.],
        [ 3.],
        [-2.]])
```

In [ ]:
```

```

General Solution

The original matrix equation:

$$\underbrace{\begin{bmatrix} 2 & 2 & 4 \\ 0 & 1 & -5 \\ 0 & 3 & 4 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}_{t} = \underbrace{\begin{bmatrix} 0 \\ 13 \\ 1 \end{bmatrix}}_{d}$$

The general solution to such a matrix equation is

$$\begin{aligned} At &= d \\ A^{-1}At &= A^{-1}d \\ I\,t &= A^{-1}d \\ t &= A^{-1}d \end{aligned}$$

Example 2

$$\begin{bmatrix} 2 & 7 & -1 & 3 & 6 \\ 3 & 3 & 1 & -1 & 5 \\ 1 & -1 & -2 & -3 & -8 \\ 4 & 5 & -3 & 7 & 10 \\ -5 & 3 & -9 & -9 & 1 \end{bmatrix} \begin{bmatrix} v \\ w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 24 \\ -1 \\ -16 \\ 52 \\ 25 \end{bmatrix}$$

In [28]:
```
A2 = np.mat("2 7 -1 3 6; 3 3 1 -1 5; 1 -1 -2 -3 -8; 4 5 -3 7 10; -5 3
 -9 -9 1"); A2
```

Out[28]:
```
matrix([[ 2,  7, -1,  3,  6],
        [ 3,  3,  1, -1,  5],
        [ 1, -1, -2, -3, -8],
        [ 4,  5, -3,  7, 10],
        [-5,  3, -9, -9,  1]])
```

In [29]:
```
d2 = np.mat("24 -1 -16 52 25").T; d2
```

Out[29]:
```
matrix([[ 24],
        [ -1],
        [-16],
        [ 52],
        [ 25]])
```

In [30]:
```
t2 = A2.I * d2
```

In [31]:
```
t2
```

Out[31]:
```
matrix([[-1.00000000e+00],
        [-1.77635684e-15],
        [-5.00000000e+00],
        [ 3.00000000e+00],
        [ 2.00000000e+00]])
```

In [32]:
```
np.round( t2 ,9)
```

Out[32]:
```
array([[-1.],
       [-0.],
       [-5.],
       [ 3.],
       [ 2.]])
```

Check:

In [33]:
```
A2*t2
```

Out[33]:
```
matrix([[ 24.],
        [ -1.],
        [-16.],
        [ 52.],
        [ 25.]])
```

In [34]:
```
A2*t2-d2
```

Out[34]:
```
matrix([[-2.13162821e-14],
        [-8.88178420e-15],
        [ 7.10542736e-15],
        [-2.13162821e-14],
        [-7.10542736e-15]])
```

In [35]:
```
np.round( A2*t2-d2 ,9)
```

Out[35]:
```
array([[-0.],
       [-0.],
       [ 0.],
       [-0.],
       [-0.]])
```

Example 3

$$\begin{bmatrix} 1 & 0 & -5 & 6 \\ 1 & 1 & 1 & 1 \\ 3 & 0 & -5 & 8 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -1 \\ 7 \\ 7 \\ 1 \end{bmatrix}$$

In [36]:
```
A3 = np.mat("1 0 -5 6; 1 1 1 1; 3 0 -5 8; 1 -1 -1 1")
```

In [37]:
```
A3
```

Out[37]:
```
matrix([[ 1,  0, -5,  6],
        [ 1,  1,  1,  1],
        [ 3,  0, -5,  8],
        [ 1, -1, -1,  1]])
```

In [38]:
```
d3 = np.mat([-1, 7, 7, 1]).T; d3
```

Out[38]:
```
matrix([[-1],
        [ 7],
        [ 7],
        [ 1]])
```

In [39]:
```
t3 = A3.I*d3; t3
```

Out[39]:
```
matrix([[-32.],
        [  0.],
        [ 16.],
        [ 16.]])
```

In [40]:
```
A3*t3
```

Out[40]:
```
matrix([[-16.],
        [  0.],
        [-48.],
        [-32.]])
```

In [41]:
```
A3.I
```

Out[41]:
```
matrix([[-1.35107989e+16, -1.35107989e+16,  1.35107989e+16,
         -1.35107989e+16],
        [-1.35107989e+16, -1.35107989e+16,  1.35107989e+16,
         -1.35107989e+16],
        [ 1.35107989e+16,  1.35107989e+16, -1.35107989e+16,
          1.35107989e+16],
        [ 1.35107989e+16,  1.35107989e+16, -1.35107989e+16,
          1.35107989e+16]])
```

Singular (Noninvertible) Matrices

Test 1: Determinants

A square matrix is invertible (nonsingular) if and only if its **determinant** is nonzero.

In [42]:
```
import numpy.linalg
```

In [43]:
```
np.linalg.det(A3)
```

Out[43]:
```
7.401486830834343e-16
```

In [44]:
```
np.linalg.det(A2)
```

Out[44]:
```
17052.000000000007
```

Test 2: Matrix rank

A square matrix is invertible (nonsingular) if and only if it is of **full rank**.

In [45]:
```
np.linalg.matrix_rank(A3)
```

Out[45]:
```
3
```

In [46]:
```
A3.shape
```

Out[46]:
```
(4, 4)
```

In [47]:
```
np.linalg.matrix_rank(A2)
```

Out[47]:
```
5
```

In [48]:
```
A2.shape
```

Out[48]:
```
(5, 5)
```

In [49]:
```
np.rank(A3)
```

```
/home/sl/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:
1: VisibleDeprecationWarning: `rank` is deprecated; use the `ndim` at
tribute or function instead. To find the rank of a matrix see `numpy.
linalg.matrix_rank`.
  """Entry point for launching an IPython kernel.
```

Out[49]:
```
2
```

In [50]:
```
A3.shape
```

Out[50]:
```
(4, 4)
```

In [ ]:
```

```

LU Decomposition

Getting back to Example 3, how do we deal with fact that A is singular?

In [51]:
```
import scipy.linalg
```

In [52]:
```
P, L, U = scipy.linalg.lu(A3)
```

In [53]:
```
P
```

Out[53]:
```
array([[0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.]])
```

In [54]:
```
L
```

Out[54]:
```
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.33333333,  1.        ,  0.        ,  0.        ],
       [ 0.33333333,  0.        ,  1.        ,  0.        ],
       [ 0.33333333, -1.        , -1.        ,  1.        ]])
```

In [55]:
```
np.round( U ,9)
```

Out[55]:
```
array([[ 3.        ,  0.        , -5.        ,  8.        ],
       [ 0.        ,  1.        ,  2.66666667, -1.66666667],
       [ 0.        ,  0.        , -3.33333333,  3.33333333],
       [ 0.        ,  0.        ,  0.        ,  0.        ]])
```

In [56]:
```
np.asmatrix(P)*L*U
```

Out[56]:
```
matrix([[ 1.,  0., -5.,  6.],
        [ 1.,  1.,  1.,  1.],
        [ 3.,  0., -5.,  8.],
        [ 1., -1., -1.,  1.]])
```

In [ ]:
```

```

In [57]:
```
P
```

Out[57]:
```
array([[0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.]])
```

In [58]:
```
np.linalg.det(P)
```

Out[58]:
```
-1.0
```

In [59]:
```
L
```

Out[59]:
```
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.33333333,  1.        ,  0.        ,  0.        ],
       [ 0.33333333,  0.        ,  1.        ,  0.        ],
       [ 0.33333333, -1.        , -1.        ,  1.        ]])
```

In [60]:
```
np.linalg.det(L)
```
Out[60]:
```
1.0
```

In [61]:
```
np.round(U ,9)
```
Out[61]:
```
array([[ 3.        ,  0.        , -5.        ,  8.        ],
       [ 0.        ,  1.        ,  2.66666667, -1.66666667],
       [ 0.        ,  0.        , -3.33333333,  3.33333333],
       [ 0.        ,  0.        ,  0.        ,  0.        ]])
```

In [62]:
```
np.linalg.det(U)
```
Out[62]:
```
-7.401486830834343e-16
```

In [63]:
```
np.linalg.matrix_rank(U)
```
Out[63]:
```
3
```

In [64]:
```
PL = np.asmatrix(P)*L
```

In [65]:
```
PL
```
Out[65]:
```
matrix([[ 0.33333333,  0.        ,  1.        ,  0.        ],
        [ 0.33333333,  1.        ,  0.        ,  0.        ],
        [ 1.        ,  0.        ,  0.        ,  0.        ],
        [ 0.33333333, -1.        , -1.        ,  1.        ]])
```

In [66]:
```
PL*U
```
Out[66]:
```
matrix([[ 1.,  0., -5.,  6.],
        [ 1.,  1.,  1.,  1.],
        [ 3.,  0., -5.,  8.],
        [ 1., -1., -1.,  1.]])
```

In [67]:
```
np.linalg.det(PL)
```
Out[67]:
```
-1.0
```

In [68]:
```
PL.I
```
Out[68]:
```
matrix([[-0.        ,  0.        ,  1.        , -0.        ],
        [-0.        ,  1.        , -0.33333333, -0.        ],
        [ 1.        ,  0.        , -0.33333333, -0.        ],
        [ 1.        ,  1.        , -1.        ,  1.        ]])
```

In [ ]:

This suggests we can do the following:

$$\begin{aligned} At &= d \\ PLUt &= d \\ Ut &= (PL)^{-1}d \end{aligned}$$

In [69]:
```
np.round( U ,9)
```

Out[69]:
```
array([[ 3.        ,  0.        , -5.        ,  8.        ],
       [ 0.        ,  1.        ,  2.66666667, -1.66666667],
       [ 0.        ,  0.        , -3.33333333,  3.33333333],
       [ 0.        ,  0.        ,  0.        ,  0.        ]])
```

In [70]:
```
PL.I*d3
```

Out[70]:
```
matrix([[ 7.        ],
        [ 4.66666667],
        [-3.33333333],
        [ 0.        ]])
```

In [71]:
```
np.round( 3*U ,9)
```

Out[71]:
```
array([[  9.,   0., -15.,  24.],
       [  0.,   3.,   8.,  -5.],
       [  0.,   0., -10.,  10.],
       [  0.,   0.,   0.,   0.]])
```

In [72]:
```
3*PL.I*d3
```

Out[72]:
```
matrix([[ 21.],
        [ 14.],
        [-10.],
        [  0.]])
```

Recap

$$\begin{aligned} At &= d \\ PLUt &= d \\ Ut &= (PL)^{-1}d \end{aligned}$$

And in this case (optionally), we multiplied both sides by 3 to make the numbers nicer:

$$3Ut = 3(PL)^{-1}d$$

Or:

$$\begin{bmatrix} 9 & 0 & -15 & 24 \\ 0 & 3 & 8 & -5 \\ 0 & 0 & -10 & 10 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 21 \\ 14 \\ -10 \\ 0 \end{bmatrix}$$

In [ ]:
```
```

Solution to Example 3

Let z be anything and backsolve (easy because U is upper triangular):

$$\begin{aligned} -10y + 10z &= -10 \\ -y + z &= -1 \\ y &= z + 1 \end{aligned}$$

$$\begin{aligned} 3x + 8y - 5z &= 14 \\ 3x + 8(z+1) - 5z &= 14 \\ 3x &= 6 - 3z \\ x &= 2 - z \end{aligned}$$

$$\begin{aligned} 9w - 15y + 24z &= 21 \\ 9w - 15(z+1) + 24z &= 21 \\ 9w - 15z - 15 + 24z &= 21 \\ 9w &= 36 - 9z \\ w &= 4 - z \end{aligned}$$

Thus, for any number z,

$$t = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4-z \\ 2-z \\ z+1 \\ z \end{bmatrix}$$

is a solution.

In [73]:
```
def t3(z):
    return np.matrix([4-z, 2-z, z+1, z]).T
```

In [74]:
```
t3(1)
```

Out[74]:
```
matrix([[3],
        [1],
        [2],
        [1]])
```

In [75]:
```
t3(10.17)
```

Out[75]:
```
matrix([[-6.17],
        [-8.17],
        [11.17],
        [10.17]])
```

In [76]:
```
A3*t3(10.17)
```

Out[76]:
```
matrix([[-1.],
        [ 7.],
        [ 7.],
        [ 1.]])
```

In [77]:
```
A3*t3(0)
```

Out[77]:
```
matrix([[-1],
        [ 7],
        [ 7],
        [ 1]])
```

Underdetermined Systems of Equations

The last equation in the system

$$\begin{bmatrix} 9 & 0 & -15 & 24 \\ 0 & 3 & 8 & -5 \\ 0 & 0 & -10 & 10 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 21 \\ 14 \\ -10 \\ 0 \end{bmatrix}$$

can be eliminated since it doesn't convey any information (it is always true):

$$\begin{bmatrix} 9 & 0 & -15 & 24 \\ 0 & 3 & 8 & -5 \\ 0 & 0 & -10 & 10 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 21 \\ 14 \\ -10 \end{bmatrix}$$

This (and the original system) is an **underdetermined** system of linear equations. It has infinitely many solutions because there are more variables (degrees of freedom) than equations (constraints).

Inconsistent Systems of Equations

Conversely, a system of linear equations with no solutions is **inconsistent** or **overdetermined**.

Example 4

For example, here is an overdetermined system of three equations in two unknowns:

$$3x - 4y = 7$$
$$2x + 6y = 5$$
$$5x + 2y = 9$$

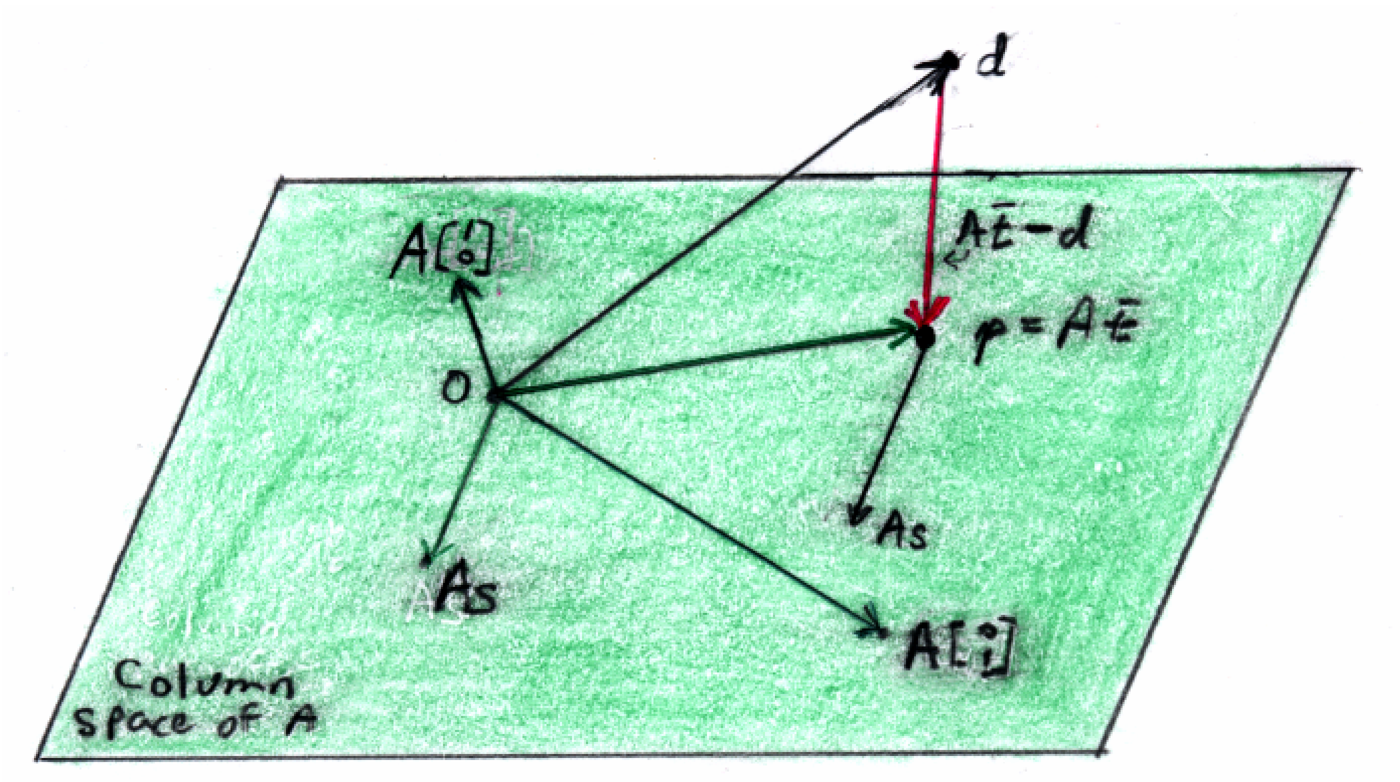
How do you "solve" such a system of equations?

In matrix notation,

$$\underset{3\times 2}{\underbrace{\begin{bmatrix}3 & -4\\ 2 & 6\\ 5 & 2\end{bmatrix}}_{A}} \quad \underset{2\times 1}{\underbrace{\begin{bmatrix}x\\ y\end{bmatrix}}_{t}} \quad = \quad \underset{3\times 1}{\underbrace{\begin{bmatrix}7\\ 5\\ 9\end{bmatrix}}_{d}}$$

Geometric Point of View

Goal: Find the projection $p = A\bar{t}$ of d onto the column space of A. It will follow that $t = \bar{t}$ minimizes the distance $\|At - d\|$ and is the **least squares** solution to the linear system $At = d$.



For all possible values of s, the vector As must be perpendicular (orthogonal) to the vector $A\bar{t} - d$:

$$\begin{aligned}(As)\cdot(A\bar{t} - d) &= 0\\ (As)^T(A\bar{t} - d) &= 0\\ s^T A^T(A\bar{t} - d) &= 0\\ s^T(A^T A\bar{t} - A^T d) &= 0\end{aligned}$$

This can only be true for *all* values of s if $A^TA\bar{t} - A^Td = 0$, or

$$A^TA\bar{t} = A^Td.$$

If A^TA is invertible, then the unique solution is

$$\bar{t} = (A^TA)^{-1}A^Td.$$

In [78]:
```
A4 = np.matrix("3 -4; 2 6; 5 2"); A4
```

Out[78]:
```
matrix([[ 3, -4],
        [ 2,  6],
        [ 5,  2]])
```

In [79]:
```
d4 = np.matrix("7; 5; 9"); d4
```

Out[79]:
```
matrix([[7],
        [5],
        [9]])
```

In [80]:
```
A4.T * A4
```

Out[80]:
```
matrix([[38, 10],
        [10, 56]])
```

Is $A_4^TA_4$ is invertible? Check the determinant:

In [81]:
```
38*56 - 10*10
```

Out[81]:
```
2028
```

In [82]:
```
np.linalg.det(A4.T*A4)
```

Out[82]:
```
2028.000000000001
```

In [83]:
```
(A4.T*A4).I * A4.T * d4
```

Out[83]:
```
matrix([[2.00000000e+00],
        [1.11022302e-16]])
```

Summary of Solution to Example 4

Thus, the least squares solution is

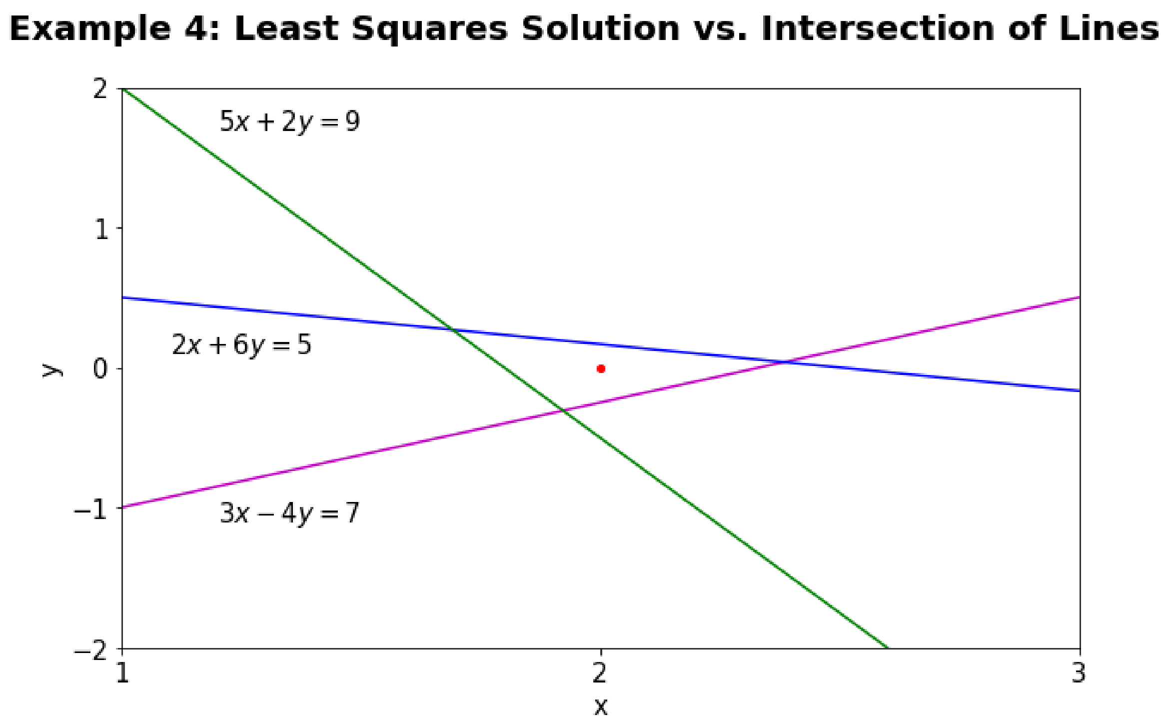
$$\bar{t} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

The projection of d onto the column space of A (the closest we could get) is

$$p = A\bar{t} = \begin{bmatrix} 3 & -4 \\ 2 & 6 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ 10 \end{bmatrix} \approx \begin{bmatrix} 7 \\ 5 \\ 9 \end{bmatrix} = d.$$

Another geometric look:

In [85]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(10,6))
plt.rcParams.update({'font.size': 15})
plt.suptitle('Example 4: Least Squares Solution vs. Intersection of L
ines', fontsize=20, fontweight='bold')
plt.xlabel('x')
plt.xticks(np.arange(1, 4, 1))
plt.xlim(1,3)
plt.yticks([-2, -1, 0, 1, 2])
plt.ylabel('y')
plt.ylim(-2,2)
x = np.arange(1, 3.1, 0.1)
plt.plot(x, 3/4*x-7/4, 'm-', x, -1/3*x+5/6, 'b-', x, -5/2*x+9/2, 'g-'
)
plt.plot(2, 0, marker='.', markersize=8, color='red')
plt.text(1.2, -1.1, r'$3x-4y=7$', color='k')
plt.text(1.1,  0.1, r'$2x+6y=5$', color='k')
plt.text(1.2,  1.7, r'$5x+2y=9$', color='k')
plt.show()
```



In general:

Least Squares Solution to a System of Equations

$$\underset{n\times k}{A} \quad \underset{k\times 1}{t} \quad = \quad \underset{n\times 1}{d}$$

The **least squares** solution $\bar{t}$ to a system $At = d$ of n linear equations in k unknowns satisfies the **normal equations**:

$$A^T A\bar{t} = A^T d.$$

If the columns of A are linearly independent, then $A^T A$ is invertible and the unique least squares solution is

$$\bar{t} = (A^T A)^{-1} A^T d.$$

Example 5

A system of five linear equations in one unknown:

$$\begin{aligned} x &= 37 \\ x &= 22 \\ x &= 70 \\ x &= 16 \\ x &= 84 \end{aligned}$$

In matrix notation,

$$\underset{5\times 1}{\begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix}} \quad \underset{1\times 1}{\begin{bmatrix}x\end{bmatrix}} \quad = \quad \underset{5\times 1}{\begin{bmatrix}37\\22\\70\\16\\84\end{bmatrix}}$$

$$\underset{5\times 1}{A} \quad \underset{1\times 1}{t} \quad \quad \underset{5\times 1}{d}$$

In [86]:
```
A5 = np.matrix("1; 1; 1; 1; 1")
```

In [87]:
```
d5 = np.matrix("37; 22; 70; 16; 84")
```

In [88]:
```
A5.T*A5
```

Out[88]:
```
matrix([[5]])
```

In [89]:
```
A5.T*d5
```

Out[89]:
```
matrix([[229]])
```

In [90]:
```
(A5.T*A5).I * A5.T * d5
```

Out[90]:
```
matrix([[45.8]])
```

Question

If you replace the equation $x = 70$ with the equivalent equation $2x = 140$, does the answer change?

In [91]:
```
A52 = np.matrix("1; 1; 2; 1; 1"); A52
```

Out[91]:
```
matrix([[1],
        [1],
        [2],
        [1],
        [1]])
```

In [92]:
```
d52 = np.matrix("37; 22; 140; 16; 84"); d52
```

Out[92]:
```
matrix([[ 37],
        [ 22],
        [140],
        [ 16],
        [ 84]])
```

In [93]:
```
(A52.T*A52).I * A52.T * d52
```

Out[93]:
```
matrix([[54.875]])
```

In [ ]:

Linear Regression

Problem: Given n data points $(x_1, y_1), \dots, (x_n, y_n)$, find the line $y = mx + b$ that best fits the data.

x is the **independent variable** and y is the **dependent variable**.

Here, b and m are the unknowns, and the x_i and y_i are known data points. Our goal is to find the best solution to the following overdetermined system of n linear equations in two unknowns (b and m):

$$b + x_1 m = y_1$$
$$\vdots$$
$$b + x_n m = y_n$$

Or, in matrix form,

$$\begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$$\underset{n\times 2}{A} \qquad \underset{2\times 1}{t} \qquad \underset{n\times 1}{Y}$$

The least squares best fit values of b and m are

$$\begin{bmatrix} b \\ m \end{bmatrix} = \bar{t} = (A^T A)^{-1} A^T Y.$$

We know $A^T A$ will be invertible if the columns of A are linearly independent.

A set of columns (vectors) is **linearly independent** if and only if there is no one column that can be expressed as a linear combination (sum of scalar multiples) of the other columns.

Intuitively, what must be true of the x_i for this to be true?

The columns of A are **not** linearly independent if

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \beta \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} \beta \\ \vdots \\ \beta \end{bmatrix}.$$

That is, the x_i's are all equal. It is not surprising that linear regression would not work if all of the values of the independent variable x in the data were the same value. Otherwise, linear regression works.

Example 6

In [94]:
```
from sklearn import datasets
```

In [95]:
```
boston = datasets.load_boston()
```

In [96]:
```
print(boston.DESCR)
```

```
Boston House Prices dataset
===========================

Notes
------
Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over
25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds
river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to
1940
        - DIS      weighted distances to five Boston employment centr
es
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of bl
acks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
http://archive.ics.uci.edu/ml/datasets/Housing


This dataset was taken from the StatLib library which is maintained a
t Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedo
nic
prices and the demand for clean air', J. Environ. Economics & Managem
ent,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression dia
gnostics
...', Wiley, 1980.   N.B. Various transformations are used in the tab
le on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning pa
pers that address regression
```

```
problems.

**References**

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Infl
uential Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Lear
ning. In Proceedings on the Tenth International Conference of Machine
Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufm
ann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Housing)
```

In [97]:
```
type(boston.data)
```

Out[97]:
```
numpy.ndarray
```

In [98]:
```
boston.data.shape
```

Out[98]:
```
(506, 13)
```

In [99]:
```
boston.feature_names
```

Out[99]:
```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RA
D',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

In [100]:
```
boston.feature_names[5]
```

Out[100]:
```
'RM'
```

In [101]:
```
Xrm = np.asmatrix(boston.data[:,5]).T; Xrm[0:10]
```

Out[101]:
```
matrix([[6.575],
        [6.421],
        [7.185],
        [6.998],
        [7.147],
        [6.43 ],
        [6.012],
        [6.172],
        [5.631],
        [6.004]])
```

In [102]:
```
Xrm.shape
```

Out[102]:
```
(506, 1)
```

In [103]:
```
A6 = np.hstack( (np.matrix(np.ones(506)).T, Xrm) ); A6
```

Out[103]:
```
matrix([[1.   , 6.575],
        [1.   , 6.421],
        [1.   , 7.185],
        ...,
        [1.   , 6.976],
        [1.   , 6.794],
        [1.   , 6.03 ]])
```

In [104]:
```
Y = np.asmatrix(boston.target).T; Y[0:10]
```

Out[104]:
```
matrix([[24. ],
        [21.6],
        [34.7],
        [33.4],
        [36.2],
        [28.7],
        [22.9],
        [27.1],
        [16.5],
        [18.9]])
```

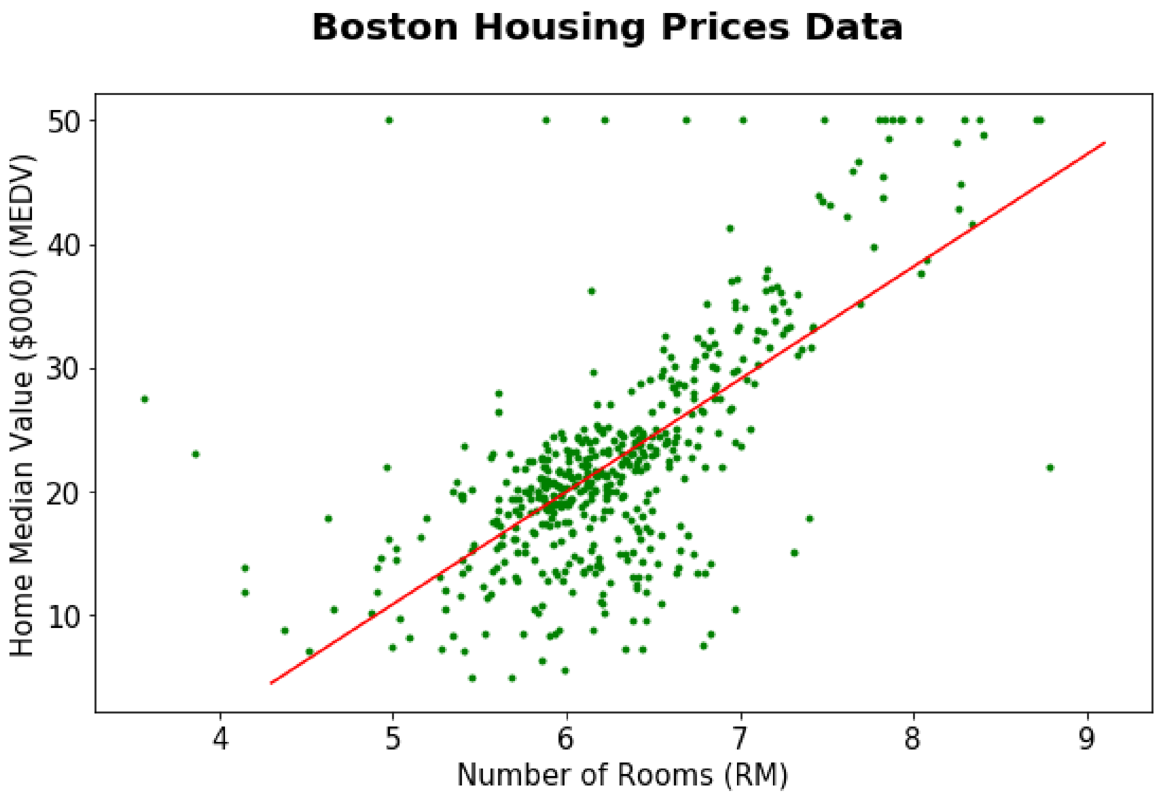
In [105]:
```
(A6.T*A6).I * A6.T * Y
```

Out[105]:
```
matrix([[-34.67062078],
        [  9.10210898]])
```

In [106]:
```
scipy.stats.linregress(boston.data[:,5], boston.target)
```

Out[106]:
```
LinregressResult(slope=9.102108981180306, intercept=-34.6706207764385
4, rvalue=0.695359947071539, pvalue=2.487228871008377e-74, stderr=0.4
1902656012134054)
```

In [107]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(10,6))
plt.rcParams.update({'font.size': 15})
plt.suptitle('Boston Housing Prices Data', fontsize=20, fontweight='b
old')
plt.xlabel('Number of Rooms (RM)')
plt.ylabel("Home Median Value ($000) (MEDV)")
plt.plot(Xrm, Y, 'g.')
x = np.array([4.3, 9.1])
plt.plot(x, 9.1021*x - 34.6706, 'r-')
plt.show()
```



[SL (11/10/2018): It was pointed out to me after the talk that the median house price is actually in multiples of $10,000, not $1,000 as indicated in the DESCR text.]

In [ ]:

Example 7—Multiple Regression

Let's add CRIM (crime rate) as a second independent variable.

In [108]:
```
boston.feature_names[0]
```

Out[108]: 'CRIM'

In [109]:
```
Xcrim = np.asmatrix(boston.data[:,0]).T; Xcrim[0:10]
```

Out[109]:
```
matrix([[0.00632],
        [0.02731],
        [0.02729],
        [0.03237],
        [0.06905],
        [0.02985],
        [0.08829],
        [0.14455],
        [0.21124],
        [0.17004]])
```

In [110]:
```
A7 = np.hstack( (A6,Xcrim) ); A7
```

Out[110]:
```
matrix([[1.0000e+00, 6.5750e+00, 6.3200e-03],
        [1.0000e+00, 6.4210e+00, 2.7310e-02],
        [1.0000e+00, 7.1850e+00, 2.7290e-02],
        ...,
        [1.0000e+00, 6.9760e+00, 6.0760e-02],
        [1.0000e+00, 6.7940e+00, 1.0959e-01],
        [1.0000e+00, 6.0300e+00, 4.7410e-02]])
```

In [111]:
```
A7.shape
```

Out[111]:
```
(506, 3)
```

In [112]:
```
np.linalg.matrix_rank(A7)
```

Out[112]:
```
3
```

In [113]:
```
np.linalg.det(A7.T*A7)
```

Out[113]:
```
4480321370.151303
```

In [114]:
```
(A7.T*A7).I * A7.T * Y
```

Out[114]:
```
matrix([[-29.30168135],
        [  8.3975317 ],
        [ -0.2618229 ]])
```

In [115]:
```
from sklearn import linear_model
```

In [116]:
```
XX = np.hstack( (Xrm,Xcrim) ); XX
```

Out[116]:
```
matrix([[6.5750e+00, 6.3200e-03],
        [6.4210e+00, 2.7310e-02],
        [7.1850e+00, 2.7290e-02],
        ...,
        [6.9760e+00, 6.0760e-02],
        [6.7940e+00, 1.0959e-01],
        [6.0300e+00, 4.7410e-02]])
```

In [117]:
```
XX.shape
```
Out[117]:
```
(506, 2)
```

In [118]:
```
regr = linear_model.LinearRegression()
```

In [119]:
```
regr.fit(XX, Y)
```
Out[119]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize
=False)
```

In [120]:
```
regr.intercept_
```
Out[120]:
```
array([-29.30168135])
```

In [121]:
```
regr.coef_
```
Out[121]:
```
array([[ 8.3975317, -0.2618229]])
```

In [ ]:

END

In [ ]:

In [ ]:

Appendix: Additional Material

[SL (11/10/2018): The following is some additional material we did not discuss during my talk today.]

In [ ]:

Appendix 1: Calculating Determinant of Example 1

In Example 1, we performed Gaussian elimination on the following matrix A. This essentially provided us with A^{-1}. It also provides an easy way to calculate the determinant of A.

$$A = \begin{bmatrix} 2 & 2 & 4 \\ 0 & 1 & -5 \\ 0 & 3 & 4 \end{bmatrix}$$

In [ ]:

First we note that the determinant respects multiplication and inverses:

$$\det A \cdot \det A^{-1} = \det(AA^{-1}) = \det I = 1 \implies \det A = \frac{1}{\det A^{-1}}.$$

The Gaussian elimination we preformed produced elementary matrices E_1, E_2, ..., E_6 such that $A^{-1} = E_6E_5E_4E_3E_2E_1$.

Thus,

$$\begin{aligned}\det A &= \left[\det A^{-1}\right]^{-1} \\ &= \left[\det(E_6E_5E_4E_3E_2E_1)\right]^{-1} \\ &= \left(\det E_6 \cdot \det E_5 \cdot \det E_4 \cdot \det E_3 \cdot \det E_2 \cdot \det E_1\right)^{-1}\end{aligned}$$

In [ ]:

We now need only figure out the determinants of the individual elementary matrices.

In [ ]:
```
E1 = scalerow(0, .5); E1
```

In [ ]:
```
E2 = addtorow(2, -3, 1); E2
```

In [ ]:
```
E3 = addtorow(0, -1, 1); E3
```

In [ ]:
```
E4 = scalerow(2, 1/19); E4
```

In [ ]:
```
E5 = addtorow(1, 5, 2); E5
```

In [ ]:
```
E6 = addtorow(0, -7, 2); E6
```

Therefore,

$$\begin{aligned}\det A &= \left(\det E_6 \cdot \det E_5 \cdot \det E_4 \cdot \det E_3 \cdot \det E_2 \cdot \det E_1\right)^{-1} \\ &= \left(1 \cdot 1 \cdot \tfrac{1}{19} \cdot 1 \cdot 1 \cdot \tfrac{1}{2}\right)^{-1} \\ &= \left(\frac{1}{38}\right)^{-1} \\ &= 38.\end{aligned}$$

In [ ]:
```
np.linalg.det(A)
```

In [ ]:

Appendix 2: Calculating Determinants Recursively

In [ ]:
```
def redet(A):
    """Determinant of matrix A.

    Recursively calculates determinant, using method typically follow
ed "by hand."
    """
    if A.shape == (1,1):
        return A[0,0]
    else:
        return sum( (-1)**j * A[0,j] * redet(np.hstack((A[1:,:j], A[1
:,j+1:])))
                   for j in range(0, A.shape[1])
                  )
```

In [ ]:
```
redet(A)
```

In [ ]:
```
np.linalg.det(A)
```

In [ ]:
```
redet(A2)
```

In [ ]:
```
np.linalg.det(A2)
```

In [ ]:
```
redet(A3)
```

In [ ]:
```
np.linalg.det(A3)
```

In [ ]:
```

```

Appendix 3: Deriving the Formulas for Linear Regression

Problem: Given n data points $(x_1, y_1), \ldots, (x_n, y_n)$, find the line $y = mx + b$ that best fits the data.

As noted previously, the least squares best fit values of b and m are given by

$$\begin{bmatrix} b \\ m \end{bmatrix} = (A^T A)^{-1} A^T Y.$$

Derivation of direct formulas of slope and intercept:

$$\begin{aligned} A^TA &= \begin{bmatrix} 1 & \cdots & 1 \\ x_1 & \cdots & x_n \end{bmatrix} \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \\ &= \begin{bmatrix} n & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 \end{bmatrix} \\ \det(A^TA) &= n\sum x_i^2 - \left(\sum x_i\right)^2 \\ (A^TA)^{-1} &= \frac{1}{n\sum x_i^2 - \left(\sum x_i\right)^2} \begin{bmatrix} \sum x_i^2 & -\sum x_i \\ -\sum x_i & n \end{bmatrix} \\ A^TY &= \begin{bmatrix} 1 & \cdots & 1 \\ x_1 & \cdots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \\ &= \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix} \\ \begin{bmatrix} b \\ m \end{bmatrix} = (A^TA)^{-1}A^TY &= \frac{1}{n\sum x_i^2 - \left(\sum x_i\right)^2} \begin{bmatrix} \left(\sum x_i^2\right)\left(\sum y_i\right) - \left(\sum x_i\right)\left(\sum x_i y_i\right) \\ n\left(\sum x_i y_i\right) - \left(\sum x_i\right)\left(\sum y_i\right) \end{bmatrix} \end{aligned}$$

Thus, the intercept and slope are given by the following formulas:

$$\begin{aligned} \text{Intercept:} \quad b &= \frac{(\sum x_i^2)(\sum y_i) - (\sum x_i)(\sum x_i y_i)}{n\sum x_i^2 - \left(\sum x_i\right)^2} \\ \text{Slope:} \quad m &= \frac{n(\sum x_i y_i) - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - \left(\sum x_i\right)^2} \end{aligned}$$

In [ ]:

Appendix 4: Condition Number

A square matrix is invertible (nonsingular) if and only if its **condition number** is finite.

Generally used in a numerical analysis context.

In [ ]:
```
np.linalg.cond(A3)
```

Very large (albeit technically finite); suggests A3 may be singular.

In [ ]:
```
np.linalg.cond(A2)
```

In [ ]:
```
```