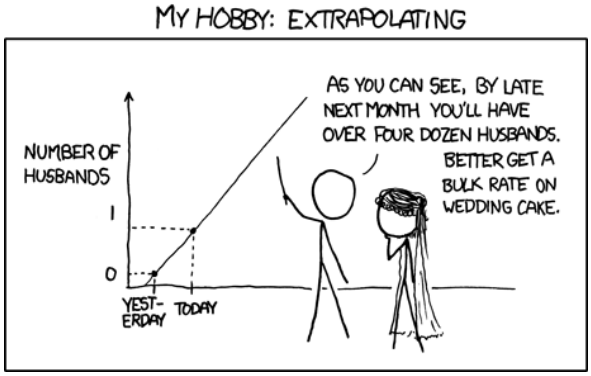
Linear Models of Data
ML_2022: Machine Learning

https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/linear_lecture/linear_lecture.pdf



A linear model of data allows us to "explain" the data and make predictions.

Linear Models

Given data `X`*, we might expect that it forms a roughly linear pattern.*

- *Two points determine a line; what if we have 100 points?*
- *To select the "best" line, we need to choose an error measure* `E`*;*
- *We may naturally regard one variable as dependent on the others, and seek a formula that predicts its value: y=b+a*x;*
- *Strictly speaking, we are seeking an* **affine** *model, because of the constant term;*
- *If we are willing to add an extra "variable" $x_0 \equiv 1$, we are seeking a true linear model, and our formulas become simpler.*
- *There are direct formulas for the solution which are useful, as long as* `n` *(number of data items) and* `d` *(number of features) are small;*
- *For large problems, an alternative approach uses* **gradient descent***;*

1 Vocabulary Lesson: "Affine" versus "Linear"

From our high school algebra days, we think of the classic linear equation as having the form:

$$y = ax + b$$

where a represents the slope, b the intercept, while x and y are the independent and dependent variables, respectively. In some cases, there might be multiple (let's say `d`) independent variables, so (using 1-based indexing for the moment) we would need to write the formula as something like

$$y = c_1 x_1 + c_2 x_2 + ... + c_d x_d + b$$

As far as mathematicians are concerned, in these two examples, y is not really a linear function of x. The test of a true linear function is that doubling the input doubles the output (or tripling, or multiplying by

any factor). This is obviously not true unless the constant b is zero. Equations of this form, in which b is not zero, are called *affine* equations.

Our work will be simplified if we can make our affine equation appear to be linear. This is easy to do, by simply inventing a new variable, x_0, with the property that it is fixed to the value 1. Now, our general equation looks like

$$y = c_0 x_0 + c_1 x_1 + c_2 x_2 + ... + c_d x_d$$

or, briefly

$$y = \sum_{i=0}^{d} c_j x_j$$

or very briefly:

$$x'c = y \qquad \text{(One Linear Equation)}$$

where the apostrophe indicates the *dot product* operation on the c and x vectors.

Now suppose that we have many (let's say n) "observations" of sets of x inputs and y outputs. We store the inputs as an $n \times d$ array X, and y is now a vector of length n. If we believe that there is an exact linear relationship between these quantities, then there must be a coefficient vector c of length d which works in every case, so that we have

$$\sum_{j=0}^{d} X_{i,j} c_j = y_i, \quad \text{for } 0 \leq i < n$$

or, as a matrix-vector statement

$$Xc = y \qquad \text{(An } n \times d \text{ system of Linear Equations)}$$

We might think that Gaussian elimination will solve this problem for us, but we're almost surely not going to be able to do so, except in the special case where $n = d$; moreover, we would have to guarantee that X is then not a singular matrix. It is more likely that $d << n$, that we have many, many observations. Linear algebra tells us that in such a case, we almost surely cannot find a coefficient vector c which makes all the linear equations exact.

2 Norm of Approximation Error

The fact that our linear system is so different from the classic case means that we will probably have to accept an approximate solution to (**??**). Obviously, if we are going to design a mathematical approach, we need to be able to define how to measure the "best" approximate solution. Just as we did for the clustering problem, we will essentially use a measure of variance as our norm.

Given an $n \times d$ matrix X, and any d-dimensional vector c, we define the `mean square error` or MSE for Equation **??** to be

$$\text{MSE}(c) = \frac{1}{n} \sum_{i=0}^{i<n} \sum_{j=0}^{j<d} (X_{i,j} c_j - y_i)^2$$

We now pose the **least squares problem**:

Problem 1 *Given the $n \times d$ matrix X and the n vector y, determine the d vector c of coefficients which achieves the minimum value of MSE(c).*

Of all d-vectors, there will turn out to be a unique c which minimizes MSE(c). This vector is known as the *least squares solution* to our problem. Now at least we know what we are hunting for, and we need to come up with a technique for finding it!

Luckily, there are several almost-equivalent methods of producing the optimal `c`.

3 Features of the sum-of-squares error E

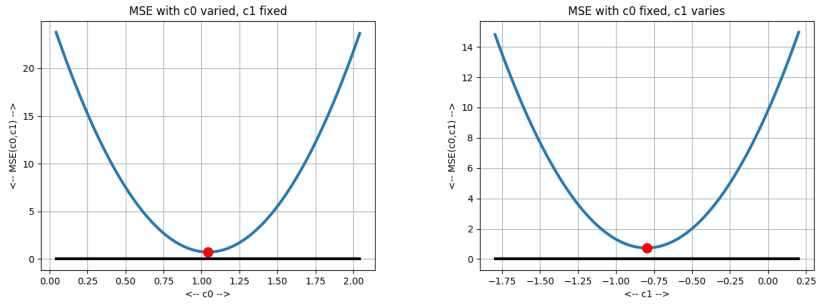
Let's return to the simplest linear model problem we have talked about, in the simple form

$$y = b + a * x$$

Suppose that we have a set of n data pairs (x_i, y_i), and that we wish to determine the optimal coefficients $c = (a, b)$ so that the resulting linear model minimizes MSE(c).

When we say that a and b are the best values, that means that choosing any other values will make MSE increase.

To verify this, let's look at what happens if we choose other values. A simple check is to hold one variable fixed, and vary the other. We get two plots, both of which look like parabolas, and both of which are minimized when we use the optimal values of c_0 and c_1:



$MSE(c_0, c_1)$ will always increase unless we use optimal values.

The fact that both graphs are parabolas is not an accident. In fact, if we think of the error as a function $E(c)$ where both c_0 and c_1 are free to vary, then a plot would display a 3D shape that looks like a cup whose bottom is at the optimal values. This suggests a way to actually solve any version of our minimization problem.

The error function $E(c)$ is a kind of parabolic shape. It is a quadratic in each of the variables c_0 and c_1. Calculus tells us that this function will be minimized when the partial derivatives $\frac{\partial E}{\partial c_0}$ and $\frac{\partial E}{\partial c_1}$ are zero. Because E is a quadratic function, after we differential we will have a linear system of equations to solve. Since we are dealing here with just two variables, we end up with a 2×2 linear system

$$E = \sum_{i=0}^{i<n} (y_i - c_0 - c_1 * x_i)^2$$

$$\frac{\partial E}{\partial c_0} = -2 * \sum_{i=0}^{i<n} y_i - c_0 - c_1 * x_i$$

$$\frac{\partial E}{\partial c_1} = -2 * \sum_{i=0}^{i<n} (y_i - c_0 - c_1 * x_i) * x_i$$

which gives us the linear system:

$$\sum_{i=1}^{n} \begin{pmatrix} 1 & x_i \\ x_i & x_i^2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \sum_{i=1}^{n} \begin{pmatrix} y_i \\ x_i y_i \end{pmatrix}$$

To simplify things, consider the following matrix:

$$A = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \ldots & \ldots \\ 1 & x_{n-1} \end{pmatrix}$$

Then our linear system actually has the form

$$A' A \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = A' y$$

This system is known as the **normal equations**. The matrix A is easy to define; the matrix $A'A$ will be square and singular. Under some simple assumptions, it will be nonsingular. If so, we can solve the linear system using a standard linear equation solver, and find our optimal coefficients c.

4 Least Squares Algorithms

Algorithm 1, the **Normal Equations** method (which we just discussed), starts with our nonrectangular $n \times d$ system

$$Xc = y$$

and multiplies by the $d \times n$ transpose of X:

$$X'Xc = X'y$$

which creates a $d \times d$ symmetric linear system. We then solve the square, and (one hopes) nonsingular linear system in the usual way.

There is a possibility that $X'X$ is singular, and there is a guarantee that the condition number of $X'X$ is larger than we would like, but in general the system is solvable just like the familiar square linear systems $A * x = b$ that we may be familiar with.

Algorithm 2 uses the **QR factorization** where Q is orthogonal and R upper triangular:

$$\begin{aligned} X = Q * R & \quad \text{Compute factorization} \\ Q * R * c = y & \quad \text{Replace } X \text{ by its factors} \\ R * c = Q' * y & \quad \text{Multiply both sides by } Q' \\ \text{solve } (R * c = Q' * y) & \quad \text{Call linear equation solver} \end{aligned}$$

The QR factorization is available in most linear algebra libraries, and so we just need to perform some matrix multiplication, and linear system solution.

Algorithm 3 uses the **SVD factorization**, which makes it possible to compute the Moore-Penrose pseudoinverse matrix P. U and V are orthogonal, and S is diagonal.

$$\begin{aligned} X = U * S * V' & \quad \text{Compute factorization} \\ P = V * \widehat{S} * U' & \quad \text{Construct pseudoinverse} \\ U * X * V' * c = y & \quad \text{Replace } X \text{ by its factors} \\ P * X * c = P * y & \quad \text{Multiply both sides by } P \\ c = P * y & \quad c \text{ is found by matrix-vector multiplication.} \end{aligned}$$

Note that the diagonal matrix $\widehat{S}$ is formed by replacing each nonzero entry $s_{i,i}$ of S by its inverse $1/s_{i,i}$, while leaving any zero diagonal elements unchanged.

The SVD factorization is available in most linear algebra libraries. If the Moore-Penrose pseudoinverse is not directly available, then the user can construct it. Once P is determined, the solution only requires a matrix-vector product.

The matrices used in Algorithms 2 and 3 have a much lower condition number than in Algorithm 1, which means the results will be more accurate. If X does not have maximum row rank, or is close to such a state, Algorithms 1 and 2 may fail or produce unsatisfactory results. Algorithm 3 will always produce a result.

5 Least Squares Solutions in Python

In Python, all three of our algorithms can be constructed by the user, or invoked by a function call. Both numpy() and scikit-learn provide additional functions which may be useful as well.

```
XTX =  np.dot ( X.T, X )
XTy = np.dot ( X.T, y )
c1 = np.linalg.solve ( XTX, XTy )
r1 = np.dot ( X, c1 ) - y
mse1 = np.sum ( r1**2 ) / n
```

Listing 1: Algorithm 1: Normal Equations

```
Q, R = np.linalg.qr ( X )
QTy = np.dot ( Q.T, y )
c2 = np.linalg.solve ( R, QTy )
r2 = np.dot ( X, c2 ) - y
mse2 = np.sum ( r2**2 ) / n
```

Listing 2: Algorith 2: QR factors

```
P = np.linalg.pinv ( X )
c3 = np.dot ( P, y )
r3 = np.dot ( X, c3 ) - y
mse3 = np.sum ( r3**2 ) / n
```

Listing 3: Algorithm 3: SVD pseudoinverse

```
c4 = np.linalg.lstsq ( X, y, rcond = None )[0]  # We only want first of four outputs.
r4 = np.dot ( X, c4 ) - y
mse4 = np.sum ( r4**2 ) / n
```

Listing 4: numpy least squares

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit ( X, y )
c5 = np.r_ [ lin_reg.intercept_, lin_reg.coef_[1:] ] # Observe underscores!
r5 = np.dot ( X, c5 ) - y
mse5 = np.sum ( r5**2 ) / n
```

Listing 5: scikit-learn

6 A "Random" Example

Let's make up a simple random example for which we know the approximate correct solution. We start with the equation

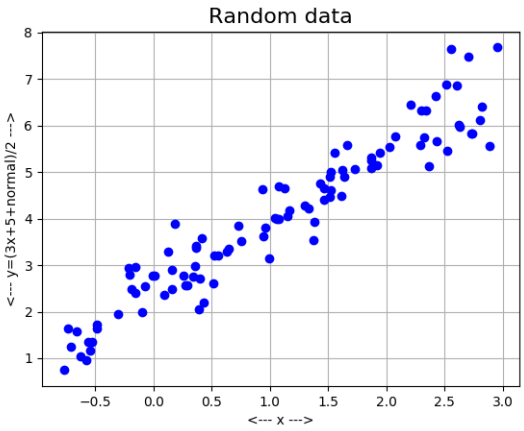
$$y = 2.5 + 1.5 * x$$

but then we add "noise" in the form of normal random values `r` with mean 0 and variance 1:

$$y = 2.5 + 1.5 * x + r$$

We set the `x` values as `n=100` uniform random values in the interval [0,10] and evaluate `y`.

We suppose the results are stored in `data`, a 100 times 2 array of `x` and `y` values. Here is a plot of our data:



A random dataset with a linear tendency.

Now, let's pretend we don't know where this data came from. By looking at the plot, we feel that there is a linear pattern to the data. The pattern is not exact, but surely there is some relationship of the form

$$y = c_0 + c_1 * x$$

that will give us a good approximation for this behavior.

To search for the linear model vector `c`, we create two new arrays:

```
X = [ np.ones(n), data[:,0] ]
y = data[:,1]
```

And now we apply our algorithms:

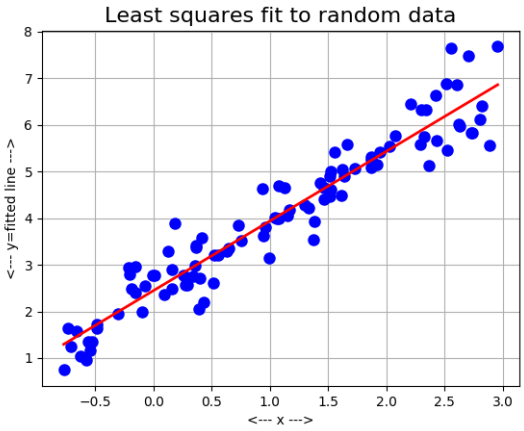
```
random_line:
  Data from random_data.txt involves n = 100 items with dimension d = 2

  Data statistics:
    Min      =  [-0.7722  0.7439]
    Max      =  [2.956 7.678]
    Range    =  [3.7282 6.9341]
    Mean     =  [1.05366843 4.020895  ]
    Variance =  [1.13735492 2.76804376]
```

```
Computed coefficient vector c:
1: Normal equations:  [2.44816256 1.49262557]
2: QR factors:        [2.44816256 1.49262557]
3: SVD pseudoinverse  [2.44816256 1.49262557]
4: numpy lstsq()      [2.44816256 1.49262557]
5: sklearn            [2.44816256 1.49262557]

MSE (mean square errors)
1: Normal equations:  0.23409536764217748
2: QR factors:        0.23409536764217745
3: SVD pseudoinverse  0.2340953676421775
4: numpy lstsq()      0.23409536764217748
5: sklearn            0.2340953676421775
```

And now we can confidently compare the fitted line $y = 2.44816 + 1.49x$ to our data $y = 2.5 + 1.5x + r$:



The least squares line fitted to the data.

7 How Much Should I Pay for a Used Ford Escort?

Suppose we want to buy a used Ford Escort. We look online, and find 23 candidates for sale, with the model year, the mileage, and the asking price. This data is now stored in the file *ford_data.txt*.

Clearly, we expect the price to be a function of both year and mileage. Moreover, the price should tend to increase with year, and to decrease with mileage. We would like to estimate this using a linear relationship. Since there are two independent variables, the result would be a plane in *(year,mileage,price)* space, so it won't be so easy to look at our results. However, assuming we seek a linear relationship

$$\text{price} = c_0 * 1 + c_1 * \text{year} + c_2 * \text{mileage}$$

which approximates the actual data, we can proceed just as we did for the random data a moment ago.

We begin by reading the data and gathering some statistics:

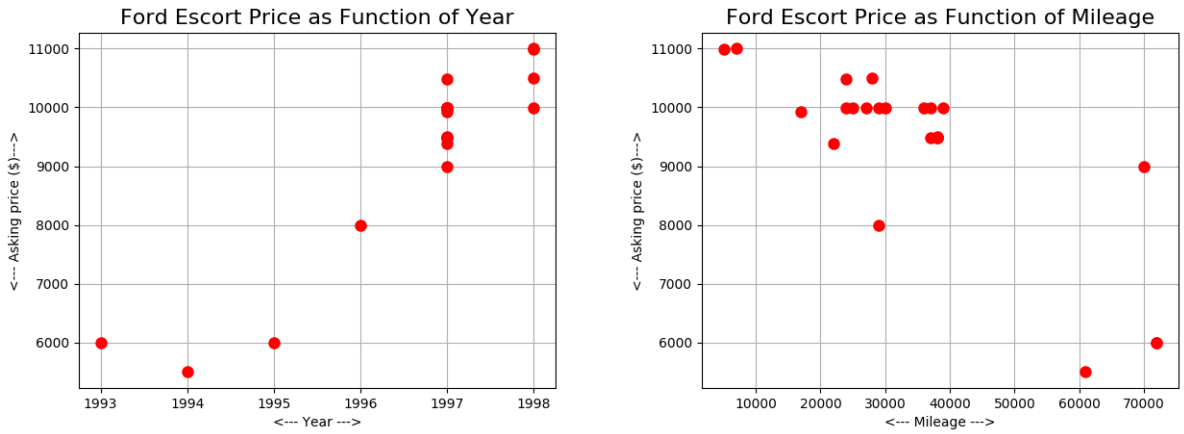
```
filename = 'ford_data.txt'
data = np.loadtxt ( filename )
n, d = np.shape ( data )
print ( "  Data from " + filename + " involves n =", n, "items with dimension d =", d )
```

```
print ( "  Data statistics:" )
print ( "    Min      = ", np.min ( data, axis = 0 ) )
print ( "    Max      = ", np.max ( data, axis = 0 ) )
print ( "    Range    = ", np.max ( data, axis = 0 ) - np.min ( data, axis = 0 ) )
print ( "    Mean     = ", np.mean ( data, axis = 0 ) )
print ( "    Variance = ", np.var ( data, axis = 0 ) )
```

Listing 6: Get data and describe it.

```
Data from ford_data.txt involves n = 23 items with dimension d = 3

Data statistics:
  Min      =  [1993. 5000. 5500.]
  Max      =  [ 1998. 72000. 11000.]
  Range    =  [5.0e+00 6.7e+04 5.5e+03]
  Mean     =  [ 1996.73913043 35000.          9332.26086957]
  Variance =  [1.41020794e+00 3.22608696e+08 2.21848671e+06]
```



Price versus year, price versus mileage.

Now we need to construct the matrix X (1,year,mileage) and the vector y (price):

```
X = np.insert ( data[0:n,0:d-1], 0, np.ones( n ), axis = 1 )
y = data[:,d-1]
```

Listing 7: Create X and y

We could set up the normal equations and solve for c:

```
XTX = np.matmul ( np.transpose ( X ), X )
XTy = np.matmul ( np.transpose ( X ), y )
c1 = np.linalg.solve ( XTX, XTy )
r1 = np.matmul ( X, c1 ) - y
mse1 = 1 / n * sum ( r1**2 )
```

Listing 8: Normal equations approach.

or use the numpy lstsq() function:

```
c4 = np.linalg.lstsq ( X, y, rcond = None )[0]
r4 = np.matmul ( X, c4 ) - y
mse4 = 1 / n * sum ( r4**2 )
```

Listing 9: lstsq() approach

The resulting coefficients suggest using the following linear approximation to the data:

$$\text{price} = -1851830 + 932 * \text{year} - 211 * \text{mileage}$$

As we expected, a higher year value improves the price, and a higher mileage reduces it.

If we plug in one of our given data values, we can actually compare the linear approximation, and decide whether the actual price seems "out of line" so to speak! For instance, let's look at our 7th and 20th data items, taking the dot product of $[c_0, c_1, c_2]$ with $[1, \text{year}_7, \text{mileage}_7]$ and with $[1, \text{year}_20, \text{mileage}_20]$:

```
                          predicted
     year  mileage  price  price      comment
 #7  1997    24000  10490   9808      $250 high?
#20  1995    72000   5994   8345      $350 low?
```

Thus, a linear model allows us to combine the independent variables(year, mileage) to predict the dependent value (price), and thus identify "outliers", that is, data items that seem far above or below the predicted value.

The same model can be used to decide a reasonable asking price if we find we have to sell our own Ford Escort, model year 1996, mileage 30,000. The model returns a price of $8,748, so make it $9,000 and be prepared to bargan down a bit.

8 Sponsorship of Basketball Players

The file *basketball_data.txt* contains 30 records. Each record stores values of 5 quantities for a player: number, height (centimeters), weight(pounds), sponsorship($), and age(years). It seems like young players are getting the most sponsorship dollars. To see if this is true, let's explore the possibility of a linear relationship of the form:

$$sponsorship = c_0 + c_1 * age$$

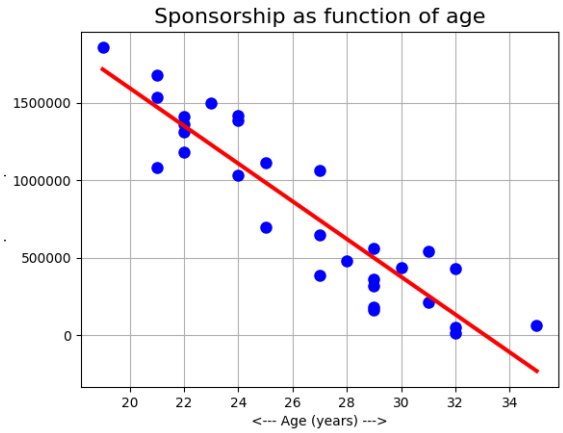
Download the data file *basketball_data.txt* and then try the following commands:

```
1   import numpy as np
2
3   data = np.loadtxt ( 'basketball_data.txt' )
4   y = data[:,3]
5   x = data[:,4]
6   n = len ( y )
7
8   X = np.zeros ( [ n, 2 ] )
9   X[:,0] = 1;
10  X[:,1] = x;
11
12  c4 = np.linalg.lstsq ( X, y, rcond = None )[0]
13  r4 = y - c4[0] - c4[1] * x
14  mse4 = 1 / n * np.sum ( r4 )**2
```

Listing 10: Set up for basketball data problem

The results of our computation suggest that every additional year of age costs a team player $120,000 in sponsorship and (theoretically) a baby (age=0) would start off at a salary of $4,000,000!

```
c[0] = 4.02735e+06
c[1] = -121703
MSE is 4.72859e+10
```



Sponsorship($) versus age (years) for 30 basketball players.

9 A multi-variable real estate problem

Let's look at a problem in which a variable y clearly depends on more than 1 or 2 variables x. In a sense, we are flying "blind", because it will not be so easy to plot our linear model against the data, since it will be a plane in a high dimensional space. This means we have to rely on our experience with low dimensional examples, and our judgement on looking at whether the resulting coefficients seem to make some sense.

The file *homes_data.txt* stores records of home sales. Each record stores 9 separate variables.

- 0: **sel**, the selling price ($)
- 1: **ask**, asking price ($)
- 2: **liv**, living area (square feet)
- 3: **rms**, rooms (#)
- 4: **bed**, bedrooms (#)
- 5: **bat**, bathrooms (#)
- 6: **age**, age (years)
- 7: **lot**, lot size (acres)
- 8: **tax**, taxes ($)

We'd like to be able to predict the selling price based on the values of `liv`, `rms`, `bed`, `bat`, `age`, `lot`, `tax`, that is, a formula

$$sel \approx c_0 + c_1\, liv + c_2\, rms + c_3\, bed + c_4\, bat + c_5\, age + c_6\, lot + c_7\, tax$$

Setting up and solving this problem is similar to what we have done before, although we need to remember to skip the "asking price" variable.

```
filename = 'homes_data.txt'
data = np.loadtxt ( filename )
n, d = np.shape ( data )

X = np.c_ [ np.ones ( n ), data[:,2:9] ]
y = data[:,0]
```

```
c4 = np.linalg.lstsq ( X, y, rcond = None )[0]
print ( "  C = ", c4 )
r4 = np.dot ( X, c4 ) - y
mse4 = np.sum ( r4**2 ) / n
print ( "  MSE = ", mse4 )
```

Listing 11: Home sale problem

The resulting coefficients come out as follows:

```
1:    28484
liv:     44
rms:  -3098
bed:   1782
bat:   2037
age:   -327
lot:   6677
tax:     18
```

The negative coefficient for age makes sense. The situation for rooms, bedrooms and bathrooms may be a little harder to explain. However, we might note that for a given living area, the more rooms we have, the smaller they are. Also, we are counting bedrooms and bathrooms twice, once separately and once as part of the total number of rooms.

Since we did not normalize our data, the size of the individual coefficients has to adjust for the size of the corresponding data item. Thus, the lot size is in acres, and usually not much bigger than 1, while the living area is in square feet, and thus in thousands.

A friend is asking \$100,000 for their house. We ask for the important data, and are told `liv` = 2000, `rms` = 9, `bed` = 3, `bat` = 2, `age` = 50, `lot` = 0.45, `tax` = 2500.

From this information, our linear model would predict the selling price to be about \$128,000, so we would advise our friend to raise their asking price.