Predicting Murder Rates With Gradient Descent

Goal

The goal of this project is to extend the techniques we learned in the one-dimensional gradient descent example to a two-dimensional domain space, solving a *linear regression problem*. This problem is also known as *curve fitting*. As part of this lab, you will learn how to compute with vectors instead of scalars. Please use file `opt/regression.py` for function `opt/plot_regression.py` for the code that draws the trace of the minimization in action (i.e., this is the file that has all the matplotlib junk). You'll see starter (and test) files in the starterkit `opt` dir.

Discussion

Problem statement

Given training data (x_i, y_i) for $i = 1..n$ samples with dependent variable y_i, we would like to predict y for some x's not in our training set. x_i is generally a vector of independent variables but we'll use a scalar. If we assume there is a linear relationship between x and y, then we can draw a line through the data and predict future values with that line function. To do that, we need to compute the two parameters of our model: a slope and a y intercept. (We will see the model below.)

For example, if we compare the number of murders per 100,000 people in Detroit to the average hourly wage, our eyeballs easily detect a correlation. Here is data suitable to copy and paste into Python:

```
HOURLY_WAGE = [2.98, 3.09, 3.23, 3.33, 3.46, 3.6, 3.73, 2.91, 4.25, 4.47, 5.04, 5.47, 5.76]
MURDERS = [8.6, 8.9, 8.52, 8.89, 13.07, 14.57, 21.36, 28.03, 31.49, 37.39, 46.26, 47.24, 52.33]
```

and here is a scatter plot and best fit line as determined by numpy (using `np.polyfit(HOURLY_WAGE, MURDERS,1)`).
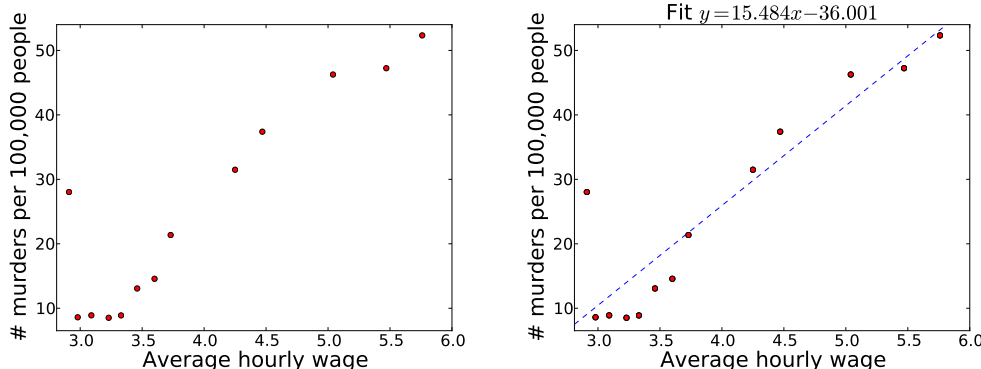
Here, for example, $x_0 = 2.98$ and $y_0 = 8.6$.

This might be a good point to remind everyone that correlation does not equal causation. I hardly think that paying people more makes them murderous, although I could see the opposite. ;) Correlation is a *necessary*

but not *sufficient* condition for causation. When you find a correlation, that gives you a candidate to check for cause-and-effect.
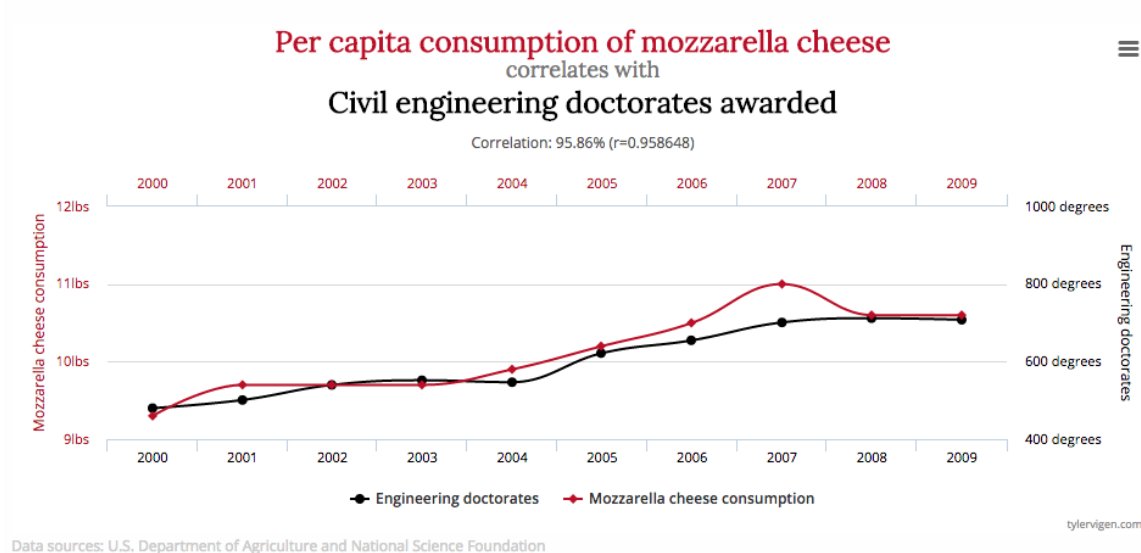
Here's another fun data set from http://www.tylervigen.com/spurious-correlations:

```
# http://www.census.gov/compendia/statab/2012/tables/12s0217.xls
MOZZ_CHEESE_PER_CAPITA_2000_2009 = [9.3, 9.7, 9.7, 9.7, 9.9, 10.2, 10.5, 11.0, 10.6, 10.6]
# http://www.nsf.gov/statistics/infbrief/nsf11305/
CIVIL_ENG_PHD_2000_2009         = [480, 501, 540, 552, 547, 622, 655, 701, 712, 708]
```


*Best fit line that minimizes squared error*

Recall the formula for a line from high school: $y = mx + b$. We normally rewrite that using elements of vector $\vec{B}$ in preparation for describing it with vector notation from linear algebra. For simplicity, though, we'll stick with scalar coefficients for now:

$$y = b_1 + b_2 x$$

The "best line" is one that minimizes some cost function that compares the known y values at x to the predicted y of the linear model that we conjure up using parameters b_1, b_2. A good measure is the *sum of squared errors*. The cost function adds up all of these squared errors to tell us how good of a fit our linear model is:

$$Cost(B) = \sum_{i=1}^{n} (\underbrace{b_1 + b_2 x_i}_{\text{linear model}} - \overbrace{y_i}^{\text{true value}})^2$$

As we change the linear model parameters, the value of the cost function will change. The following graphs shows the errors/residuals that are squared and summed to get the overall cost for two different "curve fits."
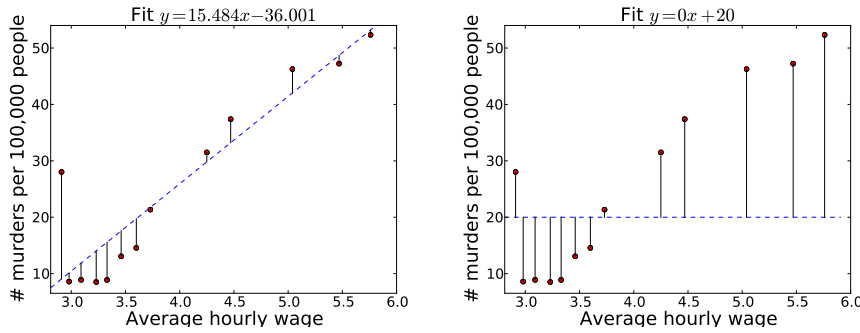
The costs are 533.82 for the left and 3563.50 for the right.

The good news is that we know the cost function is a quadratic, which is convex and has an exact solution. All we have to do is figure out where the partial derivatives of the cost function are both zero; i.e., where the cost function flattens out (at the bottom).

$$\nabla Cost(B) = 0 \qquad \text{(Analytic solution to optimization)}$$

For our purposes, though, we'll use gradient descent to minimize the cost function.

To show our prediction model in action, we can ask how many murders there would be in Detroit if the average salary were \$4.7? (Obviously, these wages are from 30 years ago.) To make a prediction, all we have to do is plug $x = 4.7$ into $y = -36.001 + 14.484x$, which gives us 32.074 murders.

Gradient descent in 3D

Before trying to minimize the cost function, it's helpful to study what the surface looks like in three dimensions, as shown in the following two graphs. The x and y dimensions are the coefficients of our linear model and the z coordinate is the cost function.
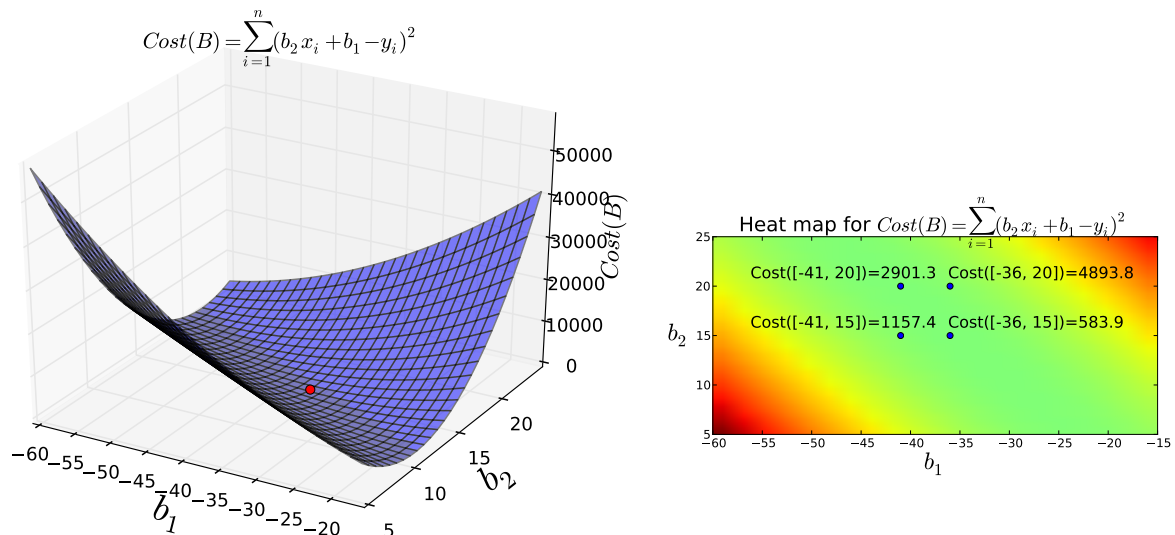
What surprised me is that changes to the slope of the linear model's coefficient b_2, away from the optimal $b_2 = 15.484$, cost much more than tweaks to the y intercept, b_1. Regardless, the surface is convex. Unfortunately, based upon the deep trough that grows slowly along the diagonal of (b_1, b_2), gradient descent takes a while to converge to the minimum. We will examine the path of gradient descent for a

few initial starting point. Wikipedia says that the Rosenbrock function is a pathological case for traditional gradient descent and it looks pretty similar to our surface with its shallow valley in this topographic view from above:
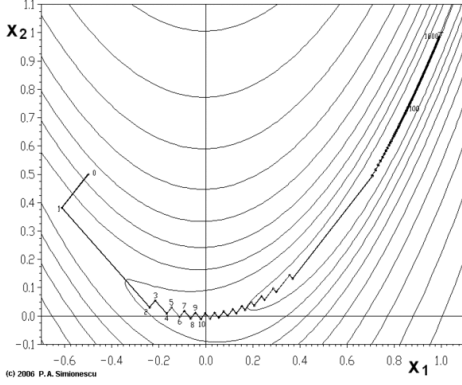
The recurrence relation for updating our estimate of $\vec{B} = [b_1, b_2]$ that minimizes $Cost(\vec{B})$ is the same as our previous lab but with a vector instead of a scalar:

$$\vec{B}_{i+1} = \vec{B}_i - \eta \nabla Cost(\vec{B}_i)$$

where we will approximate vectors of partial derivatives with partial finite differences defined generically as:

$$\nabla F(\vec{X}) = \begin{bmatrix} \frac{\partial}{x_1} F(\vec{X}) \\ \frac{\partial}{x_2} F(\vec{X}) \end{bmatrix} \approx \begin{bmatrix} \frac{F(\begin{bmatrix} x_1+h \\ x_2 \end{bmatrix}) - F(\vec{X})}{h} \\ \frac{F(\begin{bmatrix} x_1 \\ x_2+h \end{bmatrix}) - F(\vec{X})}{h} \end{bmatrix}$$

In our case, we will compute the components of a finite difference vector C' ignoring the division by the step h.

The minimization algorithm looks like:

1. Pick an initial B_0

2. Let $B = B_0$

3. $C' = \begin{bmatrix} Cost(\begin{bmatrix} b_1+h \\ b_2 \end{bmatrix}) - Cost(B_i) \\ Cost(\begin{bmatrix} b_1 \\ b_2+h \end{bmatrix}) - Cost(B_i) \end{bmatrix}$, the finite difference in each direction separately

4. Let $B_{i+1} = B_i - \eta C'$

5. Goto step 3 until $abs(Cost(B_{i+1}) - Cost(B_i)) < precision$ **and** when the $Cost$ starts going back up

Using a small learning rate, my solution takes 4843 steps starting from coordinate (-45,10) using a very small step size, which gives me a fairly decent approximation of the minimum: [-36.00066933 15.48414587]

compared to the analytic solution [-36.000625 15.484375]. Starting from about the same distance away in the shallow valley at (-45,25), my solution takes 5142 steps. Cutting my step size by 5x, takes 30463 steps but gives a slightly more precise result [-36.00063497 15.48432944]. Starting at (-45,25) takes 31958 steps. Surprisingly, when I start very close to the minimum at (-36,15), my solution takes 47350 steps and does not exactly give a more accurate result. *"Your mileage may vary."*

Trace in heat map

Trace in heat map

Trace in heat map
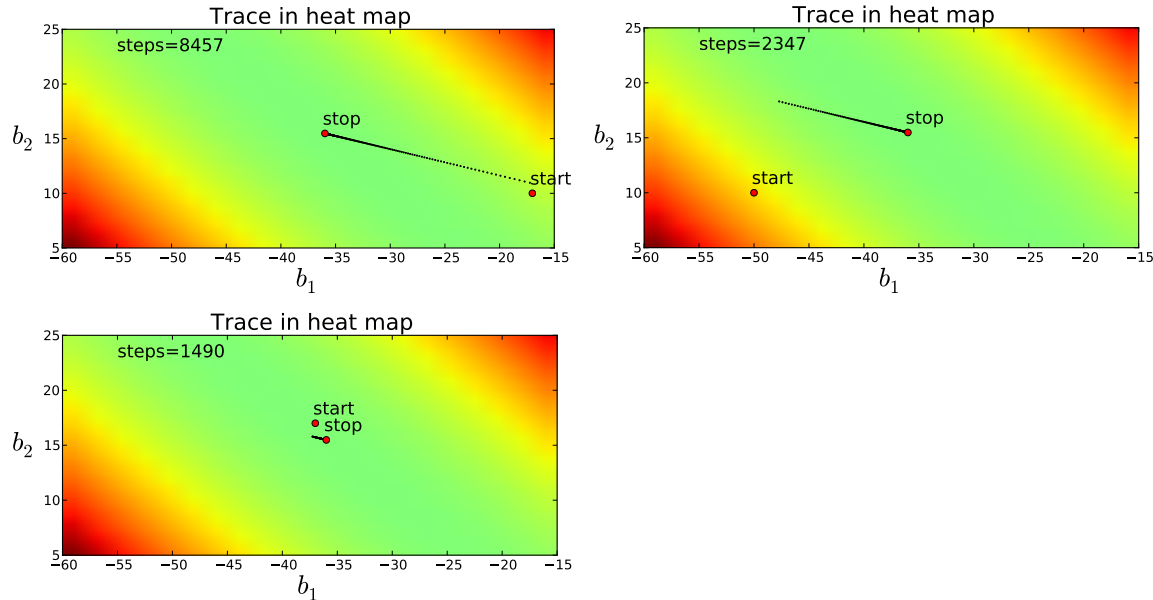
When I crank up the learning rate and use a very small step size, my solution converges must faster and with the same accuracy. For example, with 10 times the learning rate as before, (-45,25) converges in 3193 steps instead of 31958 steps.

Getting fancy

For the most part, the algorithm described above converges on the analytic solution with a high degree of precision, assuming that your h is small enough. In some cases however it only gets 3 digits of precision versus about six, depending on where the starting point is. To improve that accuracy, we can improve our termination condition from simply comparing previous and new value of the cost function to use the Euclidean distance between B_{i+1} and B_i or magnitude of the finite difference. To increase convergence rate, it's a good idea to have different η learning rates per dimension, particularly in this case because we want to speed along the valley of b_1, but not jump too far on b_2 as it is already steep; the finite difference will cause it to move along quickly. Because numpy will overload the $*$ multiple operator to work with vectors as well as scalars, η can be a 2-vector and $\eta C'$ will "just work."

For extremely large data sets or very high dimension data sets, you can use *stochastic gradient descent* which takes a single random or a small random subset of the samples when computing the cost function. Surprisingly, this still converges and takes much less time to compute. I believe that the math says that the algorithm will converge no matter how many samples (don't quote me on that). Further, picking a random subset adds noise to the process, which has been shown to pull gradient descent out of local minima and get it moving again towards a lower minimum. In our case, there is an exact solution and therefore a single minimum so we would only get a speed benefit from stochastic gradient descent.

Your task

You will use gradient descent to solve the linear regression problem above, using the same data. As part of your final submission, you must provide heat maps with traces that indicate the steps taken by your gradient descent as I have shown above. Have your program choose two random starting B_0 vectors (you can use `random.randrange()`) to produce your heat maps. In `opt/regression.py`, define a function called `minimize` that takes the indicated parameters and returns the minimum B parameters of your linear model, the number of steps, and the trace array of intermediate B_i values.

```python
def minimize(f, B0, eta, h, precision):
    trace = []
    B = B0
    steps = 0
    while True:
        steps += 1
        if steps % 10 == 0: # only capture every 10th value
            trace.append(B)
        ...
    ...
    return (B, steps, trace)
```

As an example, in my `opt/plot_regression.py` script, I call that function like this:

```python
(m,steps,trace) = minimize(Cost, B0, LEARNING_RATE, h, PRECISION)
```

Use `pylab.imshow()` to draw the heat map whose b_1 are the y intercepts, b_2 coordinates are the slopes, and heat value is the cost of x, y. Hide all of your heat map construction in function in `opt/plot_regression.py`:

```python
def heatmap(X, Y, trace): # trace is a list of [b1, b2] pairs
    ...
```

so we can call it like this:

```python
heatmap(HOURLY_WAGE, MURDERS, Cost, trace)
show() # finally show the heatmap
```

Your `heatmap` function will create two ranges for the b_1 and b_2 coordinates and then simply compute the cost, of storing it in a matrix such as `C[b1][b2] = Cost([b1,b2],...)`. The trace is plotted on top of the map once you create it. The coordinate ranges of the heat map are not related to the trace at all. It took me a while to figure out all of the crazy methods to draw the heat maps so I'll toss you a bone and just provide it for you (it goes in your `heatmap()` function):

```python
imshow(C,
        origin='lower',
        extent=[min(b1), max(b1), min(b2), max(b2)],
        vmax=abs(C).max(), vmin=-abs(C).max()
)
```

I plot the trace using:

```python
plot(p[0], p[1], "ko", markersize=1)
```

Please show the information as I have shown in the graphs to make it easier to compare results and for me to grade.

You will have to pick an appropriately small step value h to get a decent approximation of the derivative through finite differences. You want that number to be small enough that your algorithm does not oscillate around the minimum. If the number is too big it will compute a finite difference that leads to B_{i+1} leaping across the minimum to the other wall of the function. You must pick a learning rate η that allows you to go as fast as you can but not so fast that it overruns the minimum back and forth. When I crank up my learning rate too far, I also see the algorithm go off into the weeds and stops with a minimum of [-4.86000929e+10 -2.85744570e+12].

Testing

You can run opt/test_regression.py to check your miminize function. Here's my sample run:

```
$ python test_regression.py
starting at [-23, 12]
gradient descent gives [-36.00034318 15.48430235] Cost 2373170.71366
exact is [-36.000625000000007, 15.484375]
starting at [-50, 10]
gradient descent gives [-36.00062579 15.48437039] Cost 2373166.75674
exact is [-36.000625000000007, 15.484375]
starting at [-36, 17]
gradient descent gives [-36.00062582 15.4843704 ] Cost 2373166.75627
exact is [-36.000625000000007, 15.484375]
starting at [-36, 15]
gradient descent gives [-36.00034322 15.48430236] Cost 2373170.71306
exact is [-36.000625000000007, 15.484375]
starting at [100, 17]
gradient descent gives [-36.0003431 15.48430233] Cost 2373170.71481
exact is [-36.000625000000007, 15.484375]
starting at [-15, 0]
gradient descent gives [-36.00034325 15.48430237] Cost 2373170.7127
exact is [-36.000625000000007, 15.484375]
ALL TESTS PASS
```

Resources

There is a lot of material out there on the web that can be helpful.

- Finite difference at Wikipedia

- Stochastic Gradient Descent Tricks

- Numerical recipes (See Chap 10 on minimization of functions)

- Single verbal minimization in line searches

- Andrew Ng's CS229 Lecture notes

- Data analysis with Python

You must tweak the step size and other parameters so that your results agree with the first three decimal points of the analytic solution [-36.000625000000007, 15.484375]. (My solution is much better than that, except for a couple of weird starting positions where it only gets three decimal places.)

- Your script `regression/regression.py` with the `minimize()` function.

- Your script `regression/plot_regression.py`, which must have your `heatmap` function. This file imports the previous.

- A PDF of your graph with two visible traces on two heat maps or the same heat map if the traces are clear. The graph should include the start, stop location and the number of steps as I have done on mine. As part of your PDF, please indicate the *B* parameters you compute with your minimize function.