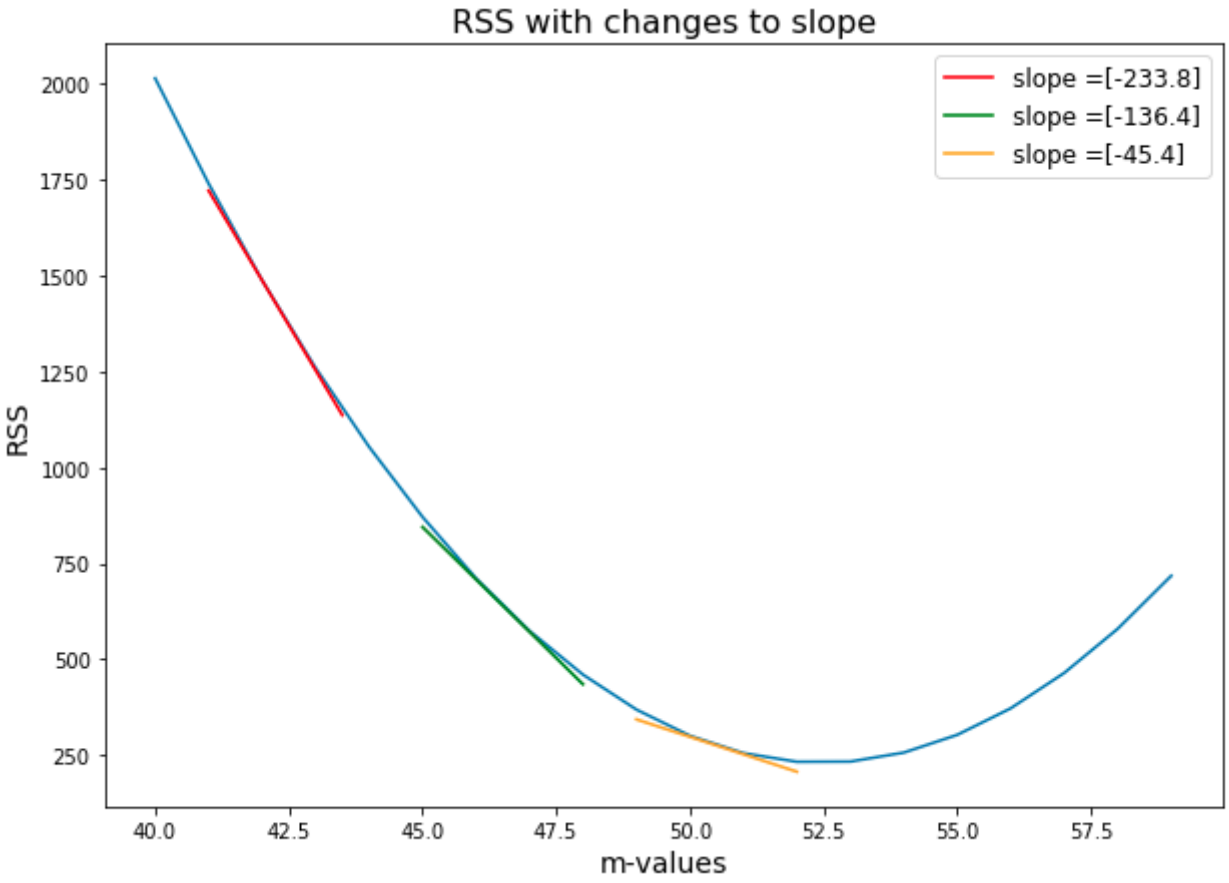
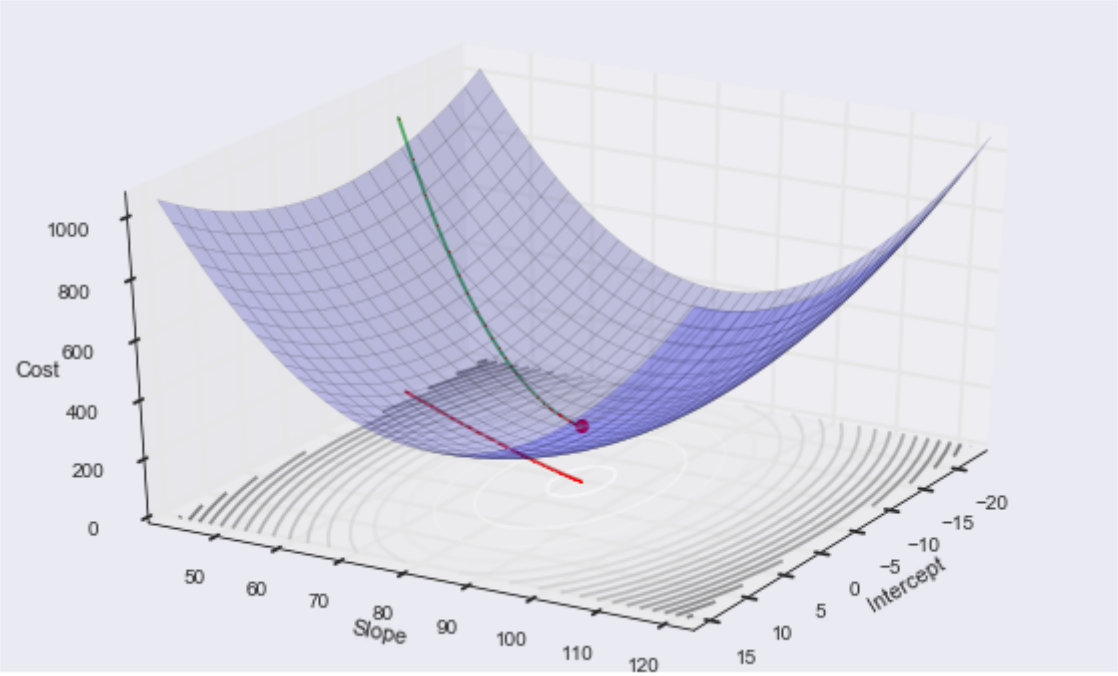
Applying Gradient Descent - Lab

Introduction

In the last lesson, we derived the functions that we help us descend along our cost functions efficiently. Remember that this technique is not so different from what we saw with using the derivative to tell us our next step size and direction in two dimensions.



When descending along our cost curve in two dimensions, we used the slope of the tangent line at each point, to tell us how large of a step to take next. And with the our cost curve being a function of m and b, we had to use the gradient to determine each step.



But really it's an analogous approach. Just like we can calculate the use derivative of a function $f(x)$ to calculate the slope at a given value of x on the graph, and thus our next step. Here, we calculated the partial derivative with respect to both variables, our slope and y-intercept, to calculate the amount to move next in either direction, and thus to steer us towards our minimum.

Objectives

You will be able to:

- Create a full gradient descent algorithm
- Apply a gradient descent algorithm on a data set with more than one variable

Reviewing our gradient descent formulas

Luckily for us, we already did the hard work of deriving these formulas. Now we get to see the fruit of our labor. The following formulas tell us how to update regression variables of m and b to approach a "best fit" line.

- $\frac{dJ}{dm}J(m,b) = -2\sum_{i=1}^{n} x_i(y_i - (mx_i + b)) = -2\sum_{i=1}^{n} x_i * \epsilon_i$
- $\frac{dJ}{db}J(m,b) = -2\sum_{i=1}^{n} (y_i - (mx_i + b)) = -2\sum_{i=1}^{n} \epsilon_i$

Now the formulas above tell us to take some dataset, with values of x and y, and then given a regression formula with values m and b, iterate through our dataset, and use the formulas to calculate an update to m and b. So ultimately, to descend along the cost function, we will use the calculations:

`current_m` = `old_m` $-(-2 * \sum_{i=1}^{n} x_i * \epsilon_i)$

`current_b` = `old_b` $-(-2 * \sum_{i=1}^{n} \epsilon_i)$

Ok let's turn this into code. First, let's initialize our data like we did before:

Notice in data x and y are flipped data = [y, x]

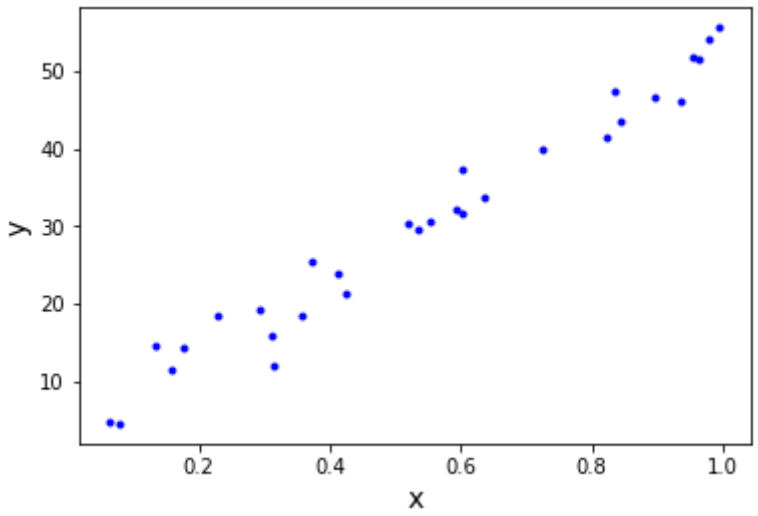
In [2]:
```
import numpy as np
np.set_printoptions(formatter={'float_kind':'{:f}'.format})
import matplotlib.pyplot as plt
np.random.seed(225)

x = np.random.rand(30, 1).reshape(30)
y_randterm = np.random.normal(0,3,30)
y = 3 + 50* x + y_randterm

data = np.array([y, x]) # data = y, x
data = np.transpose(data)


# data = [y, x1, x2, x3, ..., xn]
# y = data[0]
# x_vec = data[1::]

plt.plot(x, y, '.b')
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)
plt.show()
```



Now

- Let's set our initial regression line by initializing m and b variables as zero. Store them in `b_current` and `m_current`.
- Let's next initialize updates to these variables by setting to variables, `update_to_b` and `update_to_m` equal to 0.
- Define an `error_at` function which outputs the error ϵ_i for a given i. inputs are a row of the particular data set, b and m.
- Them, use this `error_at` function to iterate through each of the points in the dataset, and at each iteration change our `update_to_b` by $2 * \epsilon$ and change our `update_to_m` by $2 * x * \epsilon$.

In [3]:
```
# initial variables of our regression line
m_current = 0
b_current = 0

#amount to update our variables for our next step
update_to_b = 0 # set this variable, we will actually calculate it later
update_to_m = 0 # set this variable, we will actually calculate it later

# Define the error_at function
def error_at(point, m, b):
    return point[0] - (m*point[1] + b) # y - yhat

# iterate through data to change update_to_b and update_to_m
for point in data:
    # update to m = -2x*epsilon
    update_to_m += -2*point[1]*error_at(point, m_current, b_current) # -
2x(y-yhat)

    # update to b = -2*epsilon
    update_to_b += -2*error_at(point, m_current, b_current) # -2(y-yhat)

# Create new_b and new_m by subtracting the updates from the current est
imates
new_m = m_current - update_to_m
new_b = b_current - update_to_b
new_m, new_b
```

Out[3]: (1243.7171547903115, 1815.0185037502683)

In the last two lines of the code above, we calculate our `new_b` and `new_m` values by updating our taking our current values and adding our respective updates. We define a function called `error_at`, which we can use in the error component of our partial derivatives above.

The code above represents **just one** update to our regression line, and therefore just one step towards our best fit line. We'll just repeat the process to take multiple steps. But first we have to make a couple other changes.

Tweaking our approach

Ok, the above code is very close to what we want, but we just need to make tweaks to our code before it's perfect.

The first one is obvious if we think about what these formulas are really telling us to do. Look at the graph below, and think about what it means to change each of our m and b variables by at least the sum of all of the errors, of the y values that our regression line predicts and our actual data. That would be an enormous change. To ensure that we drastically updating our regression line with each step, we multiply each of these partial derivatives by a learning rate. As we have seen before, the learning rate is just a small number, like $.01$ which controls for how large our updates to the regression line will be. The learning rate is represented by the Greek letter eta, η, or alpha α. We'll use eta, so $\eta = .01$ means the learning rate is $.01$.

Multiplying our step size by our learning rate works fine, so long as we multiply both of the partial derivatives by the same amount. This is because with out gradient, $\nabla J(m, b)$, we think of as steering us in the correct direction. In other words, our derivatives ensure we are make the correct **proportional** changes to m and b. So scaling down these changes to make sure we don't update our regression line too quickly works fine, so long as we keep me moving in the correct direction. While were at it, we can also get rid of multiplying our partials by 2. As mentioned, so long as our changes are proportional we're in good shape.

For our second tweak, note that in general the larger the dataset, the larger the sum of our errors would be. But that doesn't mean our formulas are less accurate, and there deserve larger changes. It just means that the total error is larger. But we should really think accuracy as being proportional to the size of our dataset. We can correct for this effect by dividing the effect of our update by the size of our dataset, n.

Make these changes below:

In [4]:
```
#amount to update our variables for our next step
update_to_m = 0
update_to_b = 0

# define learning rate and n
learning_rate = 0.01
n = len(data)

# create update_to_b and update_to_m
for point in data:
    # update to m = -x*epsilon
    update_to_m += -point[1]*error_at(point, m_current, b_current) #remo
ve the 2 just do it to both

    # update to b = -epsilon
    update_to_b += -error_at(point, m_current, b_current) # remove the 2

# create new_b and new_m
new_m = m_current - learning_rate*update_to_m/n  # divide by n and multi
ply by learning rate
new_b = b_current - learning_rate*update_to_b/n  # divide by n and multi
ply by learning rate
new_m, new_b
```

Out[4]: (0.20728619246505192, 0.30250308395837805)

So our code now reflects what we know about our gradient descent process. Start with an initial regression line with values of m and b. Then for each point, calculate how the regression line fares against the actual point (that is, find the error). Update what our next step to the respective variable should be using by using the partial derivative. And after iterating through all of the points, update the value of b and m appropriately, scaled down by a learning rate.

Seeing our gradient descent formulas in action

As mentioned earlier, the code above represents just one update to our regression line, and therefore just one step towards our best fit line. To take multiple steps we wrap the process we want to duplicate in a function called `step_gradient` and then can call that function as much as we want. In what's next:

- Let's make sure to include a learning_rate of 0.1
- Let's output new_b and new_m as list

In [5]:
```
# I added, learning rate as a parameter so I can play around with it, bu
t it's optional
# remember points are (y, x) and not (x, y)

def step_gradient(b_current, m_current, points, learning_rate=0.1):
    # if you don't want to include as parameter, you must set it here
    # learning_rate = 0.01

    # set n
    n = len(points)

    # initalize update_to_b and update_to_m
    update_to_b, update_to_m = 0, 0

    # get update_to_ variables
    for point in points:
        # update_to_b = -epsilon
        update_to_b += -error_at(point, m_current, b_current)

        # update_to_m = -x*epsilon
        update_to_m += -point[1]*error_at(point, m_current, b_current)

    # reset m_current and b_current
    b_current -= learning_rate*update_to_b/n
    m_current -= learning_rate*update_to_m/n
    return b_current, m_current
```

Now let's initialize `b` and `m` as 0 and run a first iteration of the `step_gradient` function.

In [6]:
```
b, m = 0, 0
b, m = step_gradient(b, m, data)
b, m
# b= 3.02503, m= 2.07286
```

Out[6]: (3.0250308395837804, 2.0728619246505193)

So just looking at input and output, we begin by setting b and m to 0 amnd 0. Then from our step_gradient function, we receive new values of b and m of 3.02503 and 2.0728. Now what we need to do, is take another step in the correct direction by calling our step gradient function with our updated values of b and m.

In [41]:
```
b, m = step_gradient(b, m, data)
b,m
# b = 5.63489, m= 3.902265
```

Out[41]: (5.634896312558805, 3.9022656489039664)

Let's do this, say, 1000 times.

In [43]:
```
# create a for loop to do this
b, m = 0, 0
for step in range(1000):
    b, m = step_gradient(b, m, data)
b, m
```

Out[43]: (3.1619764855577257, 49.84313430300858)

Let's take a look at the estimates in the last iteration.

In [8]:
```
#
```

Out[8]: (3.1619764855577257, 49.84313430300858)

As you can see, our m and b values both update with each step. Not only that, but with each step, the size of the changes to m and b decrease. This is because they are approaching a best fit line.

Let's include 2 predictors, x_1 and x_2

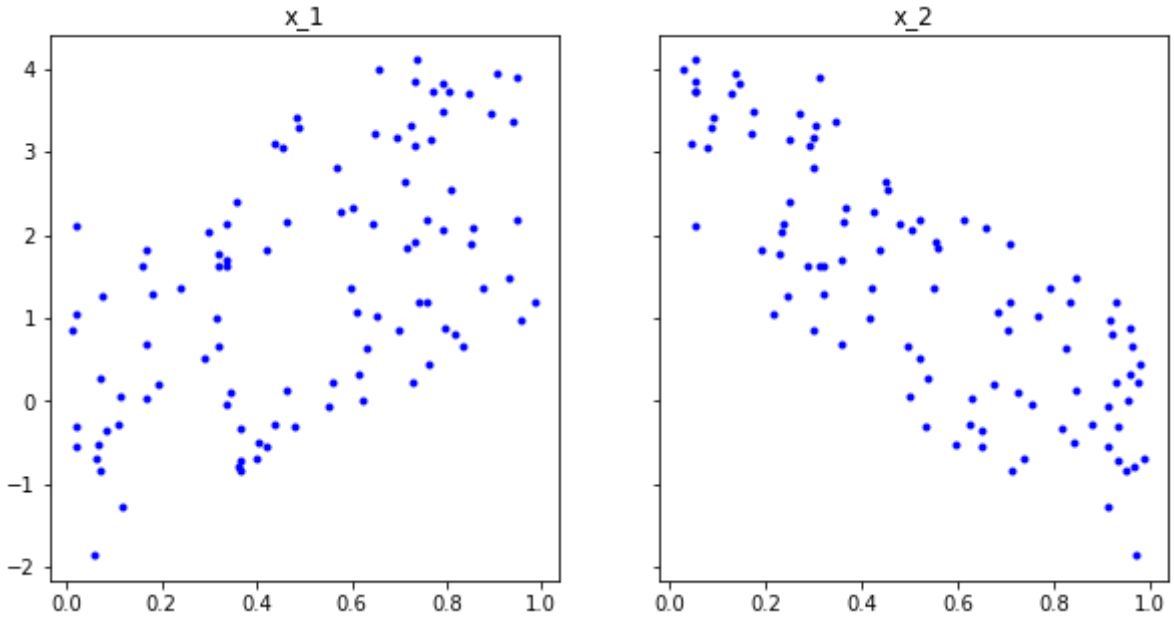
Below, we generated a problem where we have 2 predictors. We generated data such that the best fit line is around $\hat{y} = 3x_1 - 4x_2 + 2$, noting that there is random noise introduced, so the final result will never be exactly that. Let's build what we built previously, but now create a `step_gradient` function that can take an *arbitrary* number of predictors (so the function should be able to include more than 2 predictors as well). Good luck!

In [7]:
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(11)

x1 = np.random.rand(100,1).reshape(100)
x2 = np.random.rand(100,1).reshape(100)
y_randterm = np.random.normal(0,0.2,100)
y = 2+ 3* x1+ -4*x2 + y_randterm

data = np.array([y, x1, x2])
data = np.transpose(data)
```

In [8]:
```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5), sharey=True)
ax1.set_title('x_1')
ax1.plot(x1, y, '.b')
ax2.set_title('x_2')
ax2.plot(x2, y, '.b');
```



Create `step_gradient` function below

Note that, for our gradients, when having multiple predictors x_j with $j \in 1, \ldots, k$

$$\frac{dJ}{dm_j} J(m_j, b) = -2 \sum_{i=1}^{n} x_{j,i}(y_i - (\sum_{j=1}^{k} mx_{j,i} + b)) = -2 \sum_{i=1}^{n} x_{j,i} * \epsilon_i$$

$$\frac{dJ}{db} J(m_j, b) = -2 \sum_{i=1}^{n} (y_i - (\sum_{j=1}^{k} mx_{j,i} + b)) = -2 \sum_{i=1}^{n} \epsilon_i$$

So we'll have one gradient per predictor along with the gradient for the intercept!

In [9]:
```
# y - yhat (y - m1mx2 - m2x2 - b)
def general_error_at(point, m, b):
    """
    Formula for epsilon = y - (m_vec*x_vec + b)
                        = y - (m1*x1 + m2*x2 + ... mn*xn + b)
                        = y - (sum(m_vec*xvec) + b)
    Input
    points: array of points (y, x1, x2, ..., xn)
    m: current_slopes
    b: current_intercept

    return
    epsilon
    """
    epsilon = point[0] - (np.sum(m*point[1::]) + b)
    return epsilon

def step_gradient_general(b_current, m_current ,points, learning_rate=0.
1):
    # must initialize learning rate here if you didn't add as parameter
    # learning_rate = 0.1

    # initalize update_to variables
    update_to_b, update_to_m = 0, 0

    # length of points
    n = len(points)

    # points are (y, x1, x2)
    # loop to calculate updates

    for point in points:
        # update_to_b = -epsilon (from above, except removing the 2)
        y = point[0]
        x_vec = point[1::]
        update_to_b += -general_error_at(point, m, b) # -epsilon
        update_to_m += -x_vec*general_error_at(point, m, b) # -x*epsilon

    # update b_current and m_current
    b_current -= learning_rate*update_to_b/n
    m_current -= learning_rate*update_to_m/n
    return b_current, m_current
```

Apply 1 step to our data

In [10]:
```
b, m = step_gradient_general(0, 0, data)
b, m
```

Out[10]: (-0.3773393509149988, array([-0.184880, -0.253425]))

Apply 500 steps to our data

In [65]:
```
b, m = 0, 0
for step in range(500):
    b, m = step_gradient_general(b, m, data)
b, m
```

Out[65]: (1.9444283324428657, array([2.995890, -3.911055]))

Look at the last step

In [14]:
```

```

Out[14]: (1.944428332442866, array([2.995890, -3.911055]))

Level up - optional

Try your own gradient descent algorithm on the Boston Housing data set, and compare with the result from scikit learn! Be careful to test on a few continuous variables at first, and see how you perform. Scikit learn has built-in "regularization" parameters to make optimization more feasible for many parameters.

Summary

In this section, we saw our gradient descent formulas in action. The core of the gradient descent functions are understanding the two lines:

$$\frac{dJ}{dm}J(m,b) = -2\sum_{i=1}^{n} x(y_i - (mx_i + b)) = -2\sum_{i=1}^{n} x_i * \epsilon_i$$

$$\frac{dJ}{db}J(m,b) = -2\sum_{i=1}^{n} (y_i - (mx_i + b)) = -2\sum_{i=1}^{n} \epsilon_i$$

Which both look to the errors of the current regression line for our dataset to determine how to update the regression line next. These formulas came from our cost function, $J(m,b) = \sum_{i=1}^{n}(y_i - (mx_i + b))^2$, and using the gradient to find the direction of steepest descent. Translating this into code, and seeing how the regression line continued to improve in alignment with the data, we saw the effectiveness of this technique in practice. Additionally, we saw how you can extend the gradient descent algorithm to multiple predictors.