TensorFlow Regression Example

More Realistic. More data points. Batches.

The tf.estimator is for things that are easier. TensorFlow is more for things that need a specific neural network, customized, whatever...

Imports

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

import matplotlib.pyplot as plt
%matplotlib inline

import tensorflow as tf

from sklearn.model_selection import train_test_split

# remember TensorFlow and SciKit-Learn up here
```

Creating Data

One Million Points!

In [5]:
```
x_data = np.linspace(0.0, 10.0, 1000000) # We're not quite ready for a real dataset
```

In [6]:
```
x_data
```

Out[6]:
```
array([0.000000e+00, 1.000001e-05, 2.000002e-05, ..., 9.999980e+00,
       9.999990e+00, 1.000000e+01])
```

Noise

In [4]:
```
# No random seed (?)
noise = np.random.randn(len(x_data))
```

In [7]:
```
noise
```

Out[7]:
```
array([-2.53667193,  1.26634632,  2.00218565, ...,  0.58951311,
       -0.5366126 , -0.94652701])
```

Now, for the data

$y = mx + b + noise$ just to make it more difficult for the model

Jose, seemingly arbitrarily, chooses $b = 5$ and $m = 0.5$ to start

In [8]:
```
y_true = ( 0.5 * x_data ) + 5 + noise
```

Pandas

In [9]:
```
x_df = pd.DataFrame(data=x_data, columns=['X Data'])
```

In [10]:
```
x_df.head()
```

Out[10]:

	X Data
0	0.00000
1	0.00001
2	0.00002
3	0.00003
4	0.00004

In [19]:
```
y_df = pd.DataFrame(data=y_true, columns=['Y'])
```

In [20]:
```
y_df.head()
```

Out[20]:

	Y
0	2.463328
1	6.266351
2	7.002196
3	4.266070
4	4.190047

In [21]:
```
my_data = pd.concat(
         [ x_df, y_df], axis=1)
              # axis=1 keeps it from stacking on like pancakes
```

Copied from the course notes version:

```
my_data = pd.concat(
        [pd.DataFrame(data=x_data,columns=['X Data']),
         pd.DataFrame(data=y_true,columns=['Y'])
    ],
    axis=1
)
```

In [22]:
```
my_data.head()
```

Out[22]:

	X Data	Y
0	0.00000	2.463328
1	0.00001	6.266351
2	0.00002	7.002196
3	0.00003	4.266070
4	0.00004	4.190047

In [23]:
```
# my_data.plot() might crash the kernel
my_sample = my_data.sample(n=250)
```

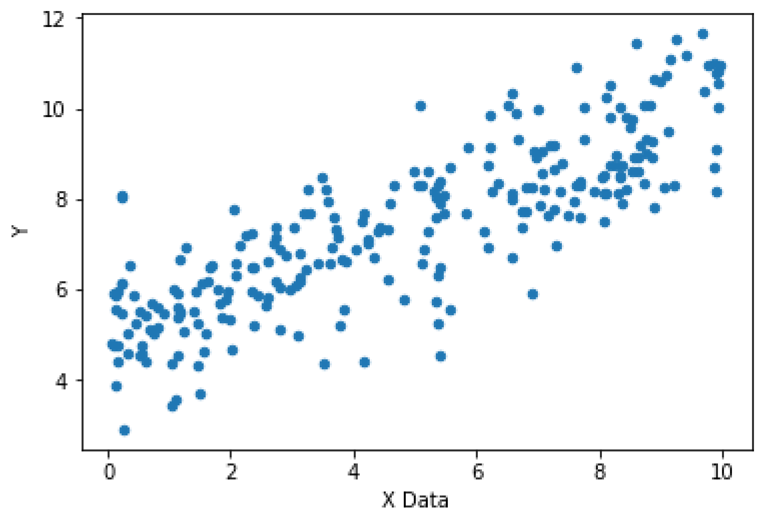
In [24]:
```
my_sample.head()
```

Out[24]:

	X Data	Y
291390	2.913903	6.751199
844584	8.445848	9.808121
853946	8.539469	9.767452
206780	2.067802	6.574018
818116	8.181168	9.789130

In [25]:
```
my_sample.plot(kind='scatter', x = 'X Data', y='Y')
```

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x21b8caee6a0>



TensorFlow

Batch Size

> We will take the data in batches (1 000 000 points is a lot to pass in at once).

In [26]:
```
# random points to grab, If you had a trillion, probably use smaller batches
batch_size = 8
```

Variables

In [34]:
```
m_pre, b_pre = np.random.randn(2)
```

In [35]:
```
type(m_pre)
```
Out[35]:
```
numpy.float64
```

In [37]:
```
type(b_pre)
```
Out[37]:
```
numpy.float64
```

In [42]:
```
# I didn't follow this, because using 'dtype' instead of 'type' worked
#tf.cast(m_pre, tf.float32)
#tf.cast(b_pre, tf.float32)
```

In [79]:
```
#  DWB, I had to add the type
#+ Jose just used, e.g. 'm = tf.Variable(0.81)'
m = tf.Variable(m_pre, dtype=tf.float32)
b = tf.Variable(b_pre, dtype=tf.float32)

print("Initally: m = " + str(m_pre) + " ; " + "b = " + str(b_pre))
```
```
Initally: m = -1.8207890158884688 ; b = 0.5308590971042547
```

Placeholders

In [50]:
```
x_ph = tf.placeholder(tf.float32, [batch_size])
```

In [51]:
```
y_ph = tf.placeholder(tf.float32, [batch_size])
```

So, I'm getting that placeholders get your data, while variables are what you're trying to predict. I'm not sure that's exactly correct, but it's what I'm getting right now.

Graph

What are we trying to do here? Fit a line to some points. So it's a $y = mx + b$ kind of graph

In [52]:
```
y_model = m * x_ph + b # Had to mess with type to get this to work
```

Loss Function

In [54]:
```
#  Remember that y_value is the true value
#+ Also, we square it to punish the error more,
#+ and thus bring it closer more quickly.
#+ could use '() ** 2' instead of tf.square()

error = tf.reduce_sum(tf.square(y_ph - y_model))
```

Optimizer

In [56]:
```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
train = optimizer.minimize(error)
```

Initialize Variables

In [57]:
```
init = tf.global_variables_initializer()
```

Session

In [61]:
```
with tf.Session() as sess:

    sess.run(init)

    batches = 1000

    for i in range(batches):

        rand_index = np.random.randint(len(x_data),
                                       size=batch_size)

        #  DWB: it seems to me we'll only we doing 8000 out
        #+ of the 1e6 points. That will make it go faster, I
        #+ guess.
        #
        #  Jose says we can play around with batches and
        #+ batch_size to see if we have enough data to
        #+ train it well. He seems to suggest that, if we
        #+ were to use more of the training data, we would
        #+ overfit to the training data. Not sure if that
        #+ applies here ... wait, yes it kinda does but not
        #+ in a way that's too concerning - we're taking
        #+ random parts ...

        feed = {x_ph:x_data[rand_index],
                y_ph:y_true[rand_index]}

        sess.run(train, feed_dict=feed)

        #  So, we have it fitting the data with 8 random points
        #+ for each

    ##endof:  for i

    #  Fetch the slope and intercept values (run will go get the
    #+ m and b placeholders)
    model_m, model_b = sess.run([m, b])
```

```
##endof:  with ... sess
```

In [62]:
```
model_m # should come out close to our 0.5
```

Out[62]:
```
0.49369615
```

In [66]:
```
model_b #should come out close to our 5
```

Out[66]:
```
4.944955
```

So, we went from whatever our original m and b values were - in my case $m = -1.8$ and $b = 0.5$. The values used for this specific training can be found with the following cell.

In [67]:
```
print("m_init = " + str(m_pre) + " ; " + "b_init = " + str(b_pre))
```

```
m_init = -1.8207890158884688 ; b_init = 0.5308590971042547
```

And we ended up with the `model_m` and `model_b` shown above, which are quite close to the values before noise, m = 0.5, b = 5; Things would look even nicer if we took the error over the value.

In [69]:
```
print("Delta_m = " + str(abs(0.5 - model_m)) + ";\n" + \
      "Delta_b = " + str(abs(5.0 - model_b)))
```

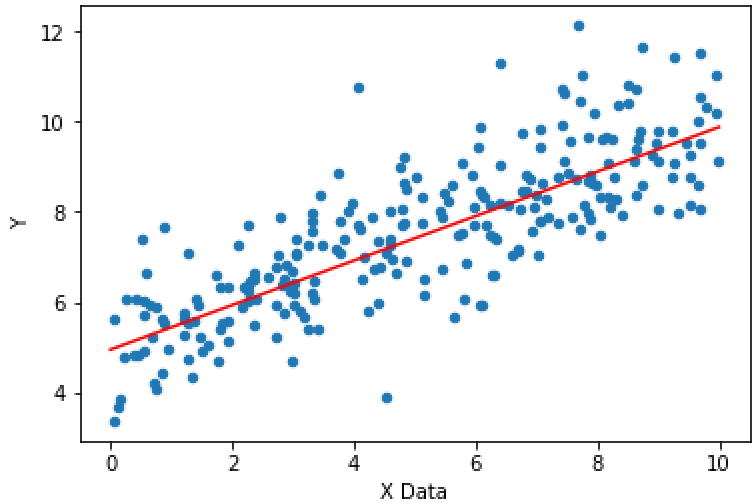
```
Delta_m = 0.006303846836090088;
Delta_b = 0.055045127868652344
```

Results

In [71]:
```
y_hat = x_data * model_m + model_b # rem. y_hat represents the predicted
```

In [73]:
```
my_data.sample(250).plot(kind="scatter",
                         x="X Data", y="Y")
plt.plot(x_data, y_hat, 'r') #  Oh, so I see Pandas is using
                             #+ the matplotlib canvas. Cool!
```

Out[73]: [<matplotlib.lines.Line2D at 0x21b8e0478d0>]



Jose changes the above code for 10k batches, I'm going to have it all re-written, so I can compare better. I will stick it to the anti-Q&R voice by not renaming the variables. Wahahaha!

I though I might have to rename them, then I think I figured that I could get rid of an error that came up by initializing the variables. Nope, had to re-put-in all the code. But I'm not renaming the variables. Wahahaha!

In [80]:
```
#  DWB, I had to add the type
#+ Jose just used, e.g. 'm = tf.Variable(0.81)'
m = tf.Variable(m_pre, dtype=tf.float32)
b = tf.Variable(b_pre, dtype=tf.float32)
print("Initally: m = " + str(m_pre) + " ; " + "b = " + str(b_pre))
x_ph = tf.placeholder(tf.float32, [batch_size])
y_ph = tf.placeholder(tf.float32, [batch_size])
y_model = m * x_ph + b
error = tf.reduce_sum(tf.square(y_ph - y_model))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
train = optimizer.minimize(error)
init = tf.global_variables_initializer()
```

```
Initally: m = -1.8207890158884688 ; b = 0.5308590971042547
```

In [81]:
```
with tf.Session() as sess:

    sess.run(init)

    batches = 10000

    for i in range(batches):

        rand_index = np.random.randint(len(x_data),
                                       size=batch_size)

        feed = {x_ph:x_data[rand_index],
                y_ph:y_true[rand_index]}

        sess.run(train, feed_dict=feed)

    ##endof:  for i

    #  Fetch the slope and intercept values (run will go get the
    #+ m and b placeholders)
    model_m, model_b = sess.run([m, b])

##endof:  with ... sess
```

In [82]:
```
model_m
```
Out[82]:
```
0.5732456
```

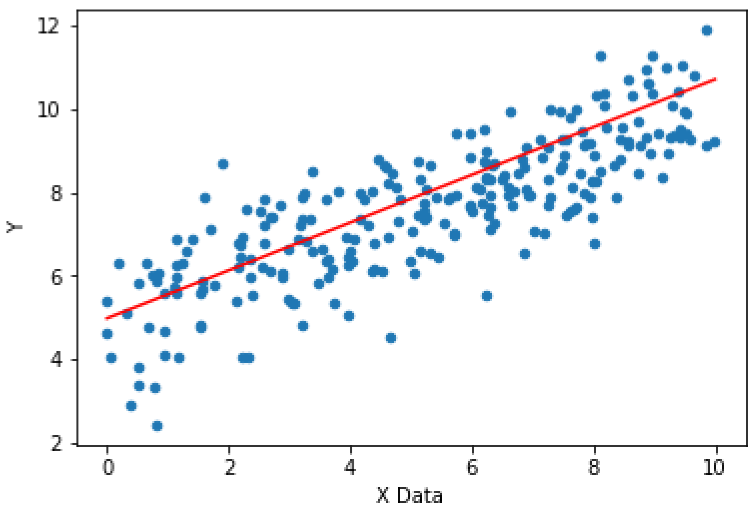
In [83]:
```
model_b
```
Out[83]:
```
4.9773397
```

After re-puttting-in the code, I got my answers.

In [85]:
```
print("m_init = " + str(m_pre) + " ; " + "b_init = " + str(b_pre))
print("m_final = " + str(model_m) + " ; " + "b_fin = " + str(model_b))

print("Delta_m = " + str(abs(0.5 - model_m)) + ";\n" + \
      "Delta_b = " + str(abs(5.0 - model_b)))
print()
print("Hmmm ...")
print()
print("Compare to:" + '\n' + \
      "Delta_m = 0.006303846836090088" + '\n' + "and" + \
      '\n' + "Delta_b = 0.055045127868652344" + '\n' +  \
      "for 8000 batches." + \
      '\n\n' + "... interesting ...")
```
```
m_init = -1.8207890158884688 ; b_init = 0.5308590971042547
m_final = 0.5732456 ; b_fin = 4.9773397
Delta_m = 0.0732455849647522;
Delta_b = 0.022660255432128906

Hmmm ...

Compare to:
Delta_m = 0.006303846836090088
and
Delta_b = 0.055045127868652344
for 8000 batches.

... interesting ...
```

In [86]:
```
y_hat = x_data * model_m + model_b
my_data.sample(250).plot(kind="scatter",
                         x="X Data", y="Y")
plt.plot(x_data, y_hat, 'r')
```

Out[86]: [<matplotlib.lines.Line2D at 0x21b8d565d68>]



Jose stated that the noise might make it so they might not be so different.

He noted (as I'd been thinking) that we haven't been doing the train/test split. We will with `tf.estimator`

tf.estimator API

> Much simpler API for basic tasks like regression! We'll talk about more abstractions like TF-Slim later on.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Train Test Split

> We haven't actually performed a train test split yet! So let's do that on our data now and perform a more realistic version of a Regression Task

In [ ]:

In [ ]:

In [ ]:

Set up Estimator Inputs

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Train the Estimator

In [ ]:

In [ ]:

Evaluation

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Predictions

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

That's all for now!