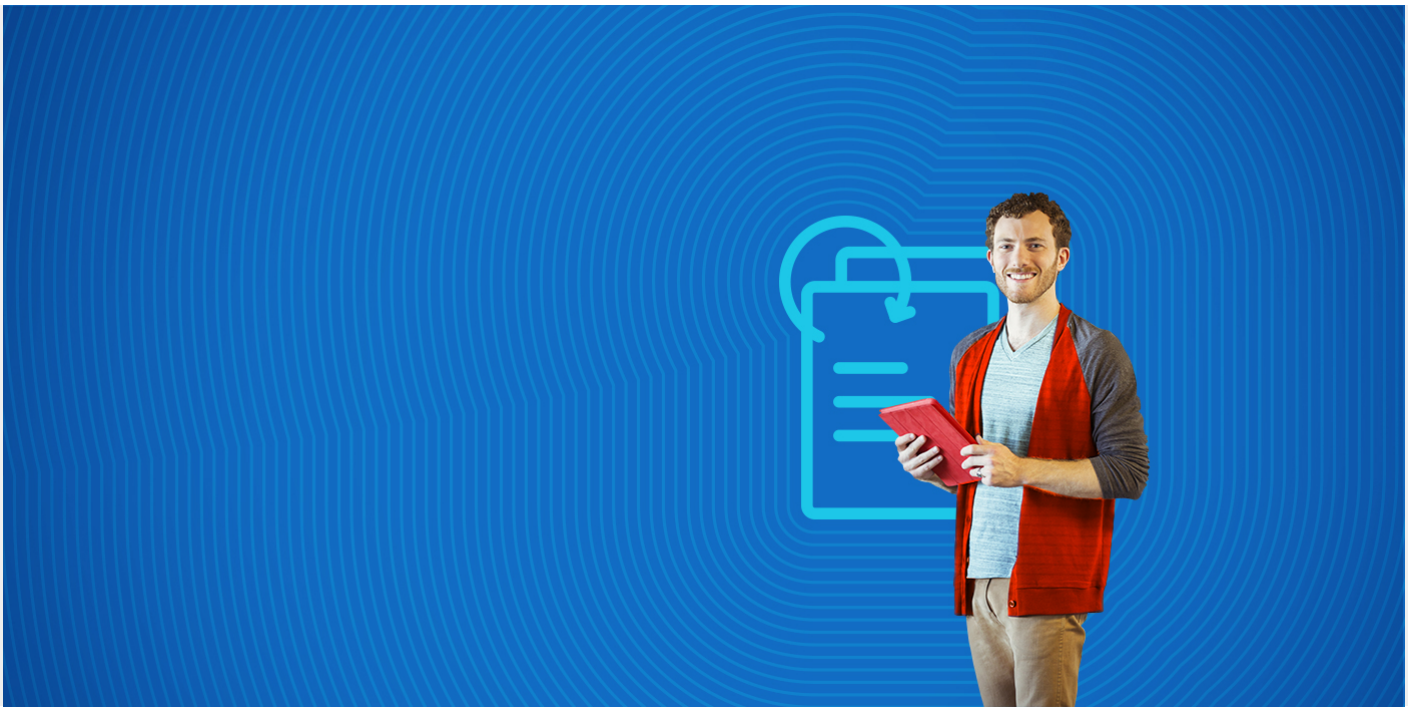
INTRODUCTION TO TENSORFLOW AND LOGISTIC REGRESSION



Here we introduce TensorFlow, an opensource machine learning library developed by Google. We explain what it does and show how to use it to do logistic regression.

Background

TensorFlow has many applications to machine learning, including neural networks. One application of neural networks is handwriting analysis. Another is facial recognition. TensorFLow is design to allow such problems to scale without limit as the nodes in the graph can be run across a distributed network. Google uses TensorFlow in some of their production applications.

One interesting aspect about TensorFlow is not only does the logic use the CPU of a machine, it can use the GPU, or graphical processor unit. That provides more power per machine as GPUs typically have a lot of power as powering the computer screen requires speed.

Install and Basic Concepts

To follow this tutorial, first install TF using the directions here.

The basis unit in TensorFlow is the **tensor**. A tensor is an array of any number of dimensions. For example:

is a 1 dimension array
] is 2 dimension array

To get started, first run Python and import TensorFlow:

```
import tensorflow as tf
```

You can assign values directly or make a **placeholder** where you assign the value later. For example a single value can be written:

```
x =  tf.constant(3.0, dtype=tf.float32)
```

Where x is an immutable **constant** (meaning you cannot change it).

But the tensor has no value until you initiate a **Session** and **run** it:

```
import tensorflow as tf
sess = tf.Session()
x =  tf.constant(3.0, dtype=tf.float32)
print(sess.run())
Outputs:
```

Or you can write:

```
import tensorflow as tf
sess = tf.Session()
y = tf.Variable(, dtype=tf.float32)
init = tf.global_variables_initializer()
sess.run(init)
print(sess.run())
Outputs:
, dtype=float32)]
```

In the example above, the **Variable**(s) have no value until you run **tf.global_variables_initializer()**.

You can add tensors and do other math, like this:

```
x =  tf.constant(, dtype=tf.float32)
y =  tf.constant(, dtype=tf.float32)
print (x + y)
print(sess.run())
outputs:
Tensor("add_4:0", shape=(2,), dtype=float32)
, dtype=float32)]
```

As you can see, the values of x and y have no value until you call **run**.

Here is another example. This is the graph of a line f(x)=mx + b, where m is the slope and b the y-intercept.

```
m = tf.Variable(, dtype=tf.float32)
b = tf.Variable(, dtype=tf.float32)
x = tf.placeholder(tf.float32)
y = m * x + b
```

You can pass an array of n values to that and run that function n times. Here we use :

```
init = tf.global_variables_initializer()
```

```
sess.run(init)
print(sess.run(y, {x: }))
Outputs:
```

Linear Regression with tf.estimator

For background on logistic regression, and interpretation of the results, you can read this document from WikiPedia. We also get our test data from that document. The goal is to predict the likelihood that a student will pass a test given how many hours they have studied.

Copy and paste the code below into the Python interpreter as we explain.

Having installed TensorFlow, now run **python**.

First we import pandas, as it is the easiest way to work with columnar data. The **hours** are floating numbers, like x.xx. We multiply them by 100 and convert them to an integer since the TensorFlow functions we used for logistic regression require either strings or integers.

```
import pandas
hours =
passx =
df = pandas.DataFrame(passx)
df = hours
df.columns =
h = df.apply(lambda x: x * 100).astype(int)
df=h
print(df)
outputs:
print(df)
hours  pass
0    0.50     0
1    0.75     0
2    1.00     0
3    1.25     0
...
```

We create a function **input_fn** that we can pass into the **LinearClassifier** model below. This function returns a data frame using the **tf.estimator.inputs.pandas_input_fn** method.

```
def input_fn(df):
    labels = df
    return tf.estimator.inputs.pandas_input_fn(
    x=df,
    y=labels,
    batch_size=100,
    num_epochs=10,
    shuffle=False,
    num_threads=5)
```

TensorFlow writes its working data to disk, so we give it a place to do that. And we have to create a

NumericColumn object, since our independent variable in continuous and not categorical. Then we create the **LinearClassifier** model.

```
import tensorflow as tf
import tempfile
model_dir = tempfile.mkdtemp()
hours = tf.feature_column.numeric_column("hours")
base_columns =
m = tf.estimator.LinearClassifier(model_dir=model_dir,
feature_columns=base_columns)
```

Now we run the **train** method.

```
m.train(input_fn(df),steps=None)
Outputs:
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Saving checkpoints for 1 into /tmp/tmpS8OD2H/model.ckpt.
INFO:tensorflow:loss = 69.3147, step = 1
INFO:tensorflow:Saving checkpoints for 10 into /tmp/tmpS8OD2H/model.ckpt.
INFO:tensorflow:Loss for final step: 54.1885.
<tensorflow.python.estimator.canned.linear.LinearClassifier object at
0x7f103b560390>
```

Use same data for test data set as the training set. In real life you would split them in two. But we have very little data here.

```
results = m.evaluate(input_fn(df),steps=None)
Outputs:
INFO:tensorflow:Starting evaluation at 2017-11-02-14:20:16
INFO:tensorflow:Restoring parameters from /tmp/tmpS8OD2H/model.ckpt-10
INFO:tensorflow:Finished evaluation at 2017-11-02-14:20:16
INFO:tensorflow:Saving dict for global step 10: accuracy = 0.75,
accuracy_baseline = 0.5, auc = 0.895, auc_precision_recall = 0.907308,
average_loss = 0.535767, global_step = 10, label/mean = 0.5, loss = 53.5767,
prediction/mean = 0.585759
```

Here we print out the same results as above but in an easier to read manner.

```
print("model directory = %s" % model_dir)
for key in sorted(results):
    print("%s: %s" % (key, results))
Outputs:
accuracy: 0.75
accuracy_baseline: 0.5
auc: 0.895
auc_precision_recall: 0.907308
average_loss: 0.535767
global_step: 10
label/mean: 0.5
```

```
loss: 53.5767
prediction/mean: 0.585759
```

The accuracy could be improved. You could create a larger data set and split the input data into a training and test data set. You could also adjust **num_epochs** and other values.