Data Recipes

Thursday, November 14, 2013

Linear Regression With Pig

It's a bit silly trying to motivate a discussion of linear regression. It's everywhere. Linear regression is typically the first pass step for understanding a dataset. Is there a linear relationship between my variables? Maybe. Let's try linear regression! In other, and less precise, words we just fit a damn line to it. Suffice it to say linear regression is one of those tools for data analysis that's not optional. You **must** know it.

In fact, I'm going to assume just that. That you've used linear regression in other contexts before and understand its utility. The issue at hand is how to *parallelize* linear regression. Why? Well, suppose you have billions of feature vectors in your data set, each with thousands of features (columns), and you want to use all of them because, why not? Suppose it doesn't fit on one machine. Now, there exists a project to address this specifically, vowpal wabbit, which you should most certainly check out, but that I'm not going to talk about. Instead, the idea is to use Apache Pig. The reason for implementing it with Pig, rather than using an existing tool, is mostly for illustration. Linear regression with pig brings up several design and implementation details that I believe you'll face when doing almost any reasonably useful machine learning at scale. In other words, how do I wire all this shit together?

Specifically, I'll address pig macros, python drivers, and using a whole relation as a scalar. Fun stuff.

linear regression

It's important that I do at least explain a bit of terminology so we're all together in this. So, rather than jump for the most general explanation immediately (why do people do that?) let's talk about something real. Suppose you've measured the current running through a circuit while slowly decreasing the resistance. You would expect the current to increase linearly as you decrease the current (ohms law). In other words,

$$I = \frac{V}{R}$$

To verify Ohm's Law (not a bad idea, I mean, maybe you're living in a dream world where physics is different and you want to know for certain...) you'd record the current I and the resistance R at every measurement while holding the voltage V constant. You'd then fit a line to the data or, more specifically, to $\frac{1}{R}$, and, if all went well, find that the slope of said line was equal to the voltage.

In the machine learning nomenclature the current would be called the *response* or target variable. All the responses together form a vector y called the *response vector.* The resistance would be called the *feature* or observation. And, if you recorded more than just the resistance, say, the temperature, then for every response you'd have a collection of features or a *feature vector*. All the feature vectors together form a matrix X. The goal of linear regression is to find the best set of weights w that, when used to form a linear combination of the features, creates a vector that is as close as possible to the response vector. So the problem can be phrased as an optimization problem. Here the function we'll minimize is the mean squared error (the square of the distance between the weighted features and the response).
Mathematically the squared error, for one feature vector x_i of length M, can be written as:

$$error^2 = (y_i - \sum_{j=1}^{M} w_j x_{i,j})^2$$

where $x_{i,0} = 1$ by definition.
So the mean squared error (mse), when we've got N measurements (feature vectors), is then:

$$mse(w) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \sum_{j=1}^{M} w_j x_{i,j})^2$$

So now the question is, exactly how are we going to minimize the mse by varying the weights? Well, it turns out there's the method called *gradient descent.* That is, the mse decreases fastest if we start with a given set of weights and travel in the direction of the negative gradient of the mse for those weights. In other words:

$$w_{new} = w - \alpha \nabla mse(w)$$

Where α is the magnitude of the step size. What this gives us is a way to update the weights until w_{new} doesn't really change much. Once the weights converge we're done.

algorithm

Alright, now that we've got a rule for updating weights, we can write down the algorithm.

1. Initialize the weights, one per feature, randomly
2. Update the weights by subtracting the gradient of the mse
3. Repeat 2 until converged

implementation

Ok great. Let's get the pieces together.

pig

Pig is going to do most of the real work. There's two main steps involved. The first step, and one that varies strongly from domain to domain, problem to problem, is loading your data and transforming it into something that a generic algorithm can handle. The second, and more interesting, is the implementation of gradient descent of the mse itself.

Here's the implementation of the gradient descent portion. I'll go over each relation in detail.

```
/*
 * Run one iteration of gradient descent for the given weights and features
 * with step size alpha. Returns the updated weights.
 *
 * features - Relation with the following schema:
 *  {response:double, vector:tuple(f0:double,f1:double,...,fN:double)}
 *
 * w - Relation with **exactly one tuple** with the following schema:
 *  {weights:tuple(w0:double,w1:double,...,wN:double)}
 *
 * alpha - Step size. Schema:
 *   alpha:double
 */
define sounder_lm_gradient_descent(features, w, alpha) returns new_weights {

  --
  -- Use a scalar cast to zip weights with their appropriate features
  --
  weighted = foreach $features {
               zipped = Zip($w.weights, vector);
               generate
                 response as response,
                 vector   as vector,
                 zipped   as zipped:bag{t:tuple(weight:double, feature:double, dimension:int)};
             };


  --
  -- Compute dot product of weights with feature vectors. First part of
  -- step adjustment.
  --
  dot_prod = foreach weighted {
               dots        = foreach zipped generate weight*feature as product;
               dot_product = SUM(dots.product);
               diff        = (dot_product-response);

               generate
                 flatten(zipped) as (weight, feature, dimension), diff as diff;
             };


  scaled = foreach dot_prod generate dimension, weight, feature*diff as feature_diff;

  --
  -- Compute step diff along each dimension. Uses combiners
  --
  steps = foreach (group scaled by (dimension,weight)) {
            factor      = ($alpha/(double)COUNT(scaled));
            weight_step = factor*SUM(scaled.feature_diff);
            new_weight  = group.weight - weight_step;
            generate
              group.dimension as dimension,
              new_weight      as weight;
          };

  --
  -- A group all is acceptable here since the previous step reduces the
  -- size down to the number of features.
  --
  $new_weights = foreach (group steps all) {
                   in_order = order steps by dimension asc;
                   as_tuple = BagToTuple(in_order.weight);
                   generate
```

```
64                      as_tuple;
65                  };
66  };
```

linear_model.pig hosted with ♥ by GitHub view raw

Go ahead and save that in a directory called 'macros'. Since gradient descent of the mean squared error for the purposes of creating a linear model is a generic problem, it makes sense to implement it as a pig macro.

In lines 19-25 we're attaching the weights to every feature vector. The Zip UDF, which can be found on github, receives the weights as a tuple and the feature vector as a tuple. The output is a bag of new tuples which contains (weight,feature,dimension). Think of Zip like a zipper where it matches the weights to their corresponding features. Importantly, the dimension (index in the input tuples) is returned as well.

Something to notice about this first bit is the scalar cast. Zip receives the *entire* weights relation as a single object. Since the weights relation is only a single tuple anyway, this is great. This is a good thing. It prevents us from doing something silly like a replicated join on a constant (which works but clutters the logic) or, worse, a cross.

Next, in lines 32-39, we're computing a portion of the gradient of the mse. The reason for the Zip udf in the first step was so the nested projection and sum to compute the dot product of the weights with the features works out cleanly.

Then, on line 42, the full gradient of the mse is computed by multiplying each feature by the error. It might not seem obvious when written like that, but the partial derivatives of the mse with respect to each weight make it work out like this. How nice.

Lines 47-54 is where the action happens. By action I mean we'll actually trigger a reduce job since everything up to this point has been map only. This is where the partial derivative bits come together. That is, we're grouping by dimension and weight (which, it turns out is precisely the thing we're differentiating with respect to) and summing each feature vector's contribution to the partial derivative along that dimension. The factor bit, on line 48, is the multiplier for the gradient. It includes the normalization term (we normalize by the number of features since we're differentiating the *mean* squared error) and the step size. The result of this step is a new weight for each dimension.

An important thing to note here is that there are only a number of partitions equivalent to the number of features. What this means is that each reduce task would get a potentially *very large* amount of data. Fortunately for us COUNT and SUM are both algebraic and so Pig will use combiners, hurray!, drastically reducing the amount of data sent to each reduce task.

Finally, on lines 60-65, we reconstruct a new tuple with the new weights and return it. The schema of this tuple should be the same as the input weight tuple.

gory details

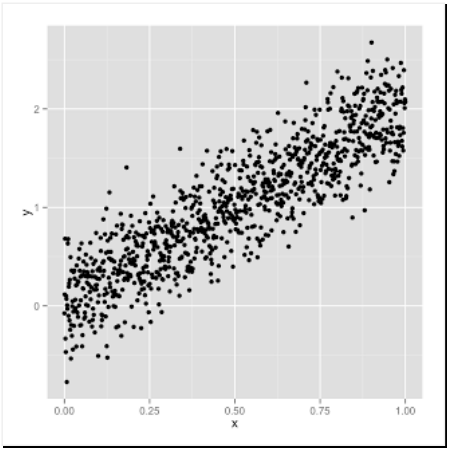
So now that we have a way to do it, we should do it. Right, I mean if we can we should. Isn't that how technology works...

I'll going to go ahead and do a completely contrived example. The reason is so that I can visualize the results.

I've created some data, called data.tsv, which satisfies the following:

$$y = 0.3r(x) + 2x$$

where $r(x)$ is a noise term. And here's the plot:



So we have two feature columns, (1.0, x), that we're trying to find the weights for. Since we've cooked this example (we already know the relationship between x and y) we expect the weights for those columns to be (0.0,2.0) if all goes well.

Now that we've got some data to work with, we'll need to write a bit more pig to load the data up and run it through our gradient descent macro.

```
1  %default STEPSIZE 0.1
2
```

```
import 'macros/linear_model.pig';

data = load '$data' as (x:double, y:double);

--
-- The weights are given a schema file by the python driver script
-- to avoid having to manually specify and arbitrarily large schema
-- with identical column schemas
--
weights = load '$input_weights' using PigStorage('\t', '-schema');


features = foreach data generate y as response, TOTUPLE(1.0,x) as vector;
weights  = foreach weights generate TOTUPLE(w0,w1) as weights;

new_weights = sounder_lm_gradient_descent(features, weights, $STEPSIZE);
new_weights = foreach new_weights generate flatten($0);

--
-- Store weights in the same way they were came in. The
-- python driver copies the input weights schema to the output
-- dir (since it's the same). The reason for this is to avoid
-- having to manually specify an as clause, in the project
-- expression above, with over 100 fields.
--
store new_weights into '$output_weights';
```

fit_line.pig hosted with ♥ by GitHub view raw

There's really not much exciting going on here. We're loading the features and weights and rearranging them to satisfy the schema that the gradient descent macro expects. The other details are related to the driver script.

The driver script is next. Since our algorithm is iterative and pig itself has no support for iteration, we're going to embed the pig script into a python (jython) program. Here's what that looks like:

```
#!/usr/bin/env python

import os
import sys
import shutil
import random
import tempfile
import Queue

from org.apache.pig.scripting import Pig
from org.codehaus.jackson.map import ObjectMapper

EPS          = 10e-6      # maximum distance between consective weights for convergence

pig_script   = sys.argv[1] # pig script to run iteratively
data_dir     = sys.argv[2] # directory where intermediate weights will be written
features     = sys.argv[3] # location, inside data_dir, where the data to fit exists
num_features = sys.argv[4] # number of features

#
# Cleanup data dir
#
cmd = "rmr %s/weight-*" % data_dir
Pig.fs(cmd)

#
# Initialize weights
#
w0_fields = []
weights   = []
for i in xrange(int(num_features)):
    weights.append(str(random.random()))
    w0_fields.append({"name":"w%s" % i,"type":25,"schema":None}) # See Pig's DataType.java

path = tempfile.mkdtemp()
w0   = open("%s/part-r-00000" % path, 'w')
w0.write("\t".join(weights)+"\n")
w0.close()

#
# Create schema for weights, place under weight-0 dir
#
w0_schema      = {"fields":w0_fields,"version":0,"sortKeys":[],"sortKeyOrders":[]}
```

```
w0_schema_file = open("%s/.pig_schema" % path, 'w')
ObjectMapper().writeValue(w0_schema_file, w0_schema);
w0_schema_file.close()

#
# Copy initial weights to fs
#
copyFromLocal = "copyFromLocal %s %s/%s" % (path, data_dir, "weight-0")
Pig.fs(copyFromLocal)



#
# Iterate until converged
#
features     = "%s/%s" % (data_dir,features)
script       = Pig.compileFromFile(pig_script)
weight_queue = Queue.Queue(25) # for moving average
avg_weight   = [0.0 for i in xrange(int(num_features))]
converged    = False
prev         = 0
weight_dir   = tempfile.mkdtemp()

while not converged:
    input_weights  = "%s/weight-%s" % (data_dir,prev)
    output_weights = "%s/weight-%s" % (data_dir,prev+1)

    bound = script.bind({'input_weights':input_weights,'output_weights':output_weights,'data':feature
    bound.runSingle()



    #
    # Copy schema for weights to each output
    #
    copyOutputSchema = "cp %s/.pig_schema %s/.pig_schema" % (input_weights, output_weights)
    Pig.fs(copyOutputSchema)

    #
    # The first few iterations the weights bounce all over the place
    #
    if (prev > 1):
        copyToLocalPrev = "copyToLocal %s/part-r-00000 %s/weight-%s" % (input_weights, weight_dir, pr
        copyToLocalNext = "copyToLocal %s/part-r-00000 %s/weight-%s" % (output_weights, weight_dir, p

        Pig.fs(copyToLocalPrev)
        Pig.fs(copyToLocalNext)

        localPrev = "%s/weight-%s" % (weight_dir, prev)
        localNext = "%s/weight-%s" % (weight_dir, prev+1)

        x1 = open(localPrev,'r').readlines()[0]
        x2 = open(localNext,'r').readlines()[0]

        x1 = [float(x.strip()) for x in x1.split("\t")]
        x2 = [float(x.strip()) for x in x2.split("\t")]

        weight_queue.put(x1)

        avg_weight = [x[1] + (x[0] - x[1])/(prev-1.0) for x in zip(x1,avg_weight)]

        #
        # Make sure to collect enough weights into the average before
        # checking for convergence
        #
        if (prev > 25):
            first_weight = weight_queue.get()
            avg_weight   = [(x[0] - x[1]/25.0 + x[2]/25.0) for x in zip(avg_weight, first_weight, x1)

            #
            # Compute distance from weight centroid to new weight
            #
            d = sum([(pair[0] - pair[1])**2 for pair in zip(x2,avg_weight)])

            converged = (d < EPS)

        os.remove(localPrev)
        os.remove(localNext)
```

```
122     prev += 1
123
124 #
125 # Cleanup
126 #
127 shutil.rmtree(path)
128 shutil.rmtree(weight_dir)
```

driver.py hosted with ♥ by GitHub view raw

There's a lot going on here (it's cluttered looking because Pig doesn't allow class or function definitions in driver scripts) that's not that interesting and pretty easy to understand so I'll just go over the high points:

- We initialize the weights randomly and write a .pig_schema file as well. The reason for writing the schema file is so that it's unnecessary to write the redundant schema for weights in the pig script itself.
- We want the driver to be agnostic to whether we're running in local mode or mapreduce mode. Thus we copy the weights to the filesystem using the *copyFromLocal* fs command. In local mode this just puts the initial weights on the local fs whereas in mapreduce mode this'll place them on the hdfs.
- Then we iterate until the convergence criteria is met. Each iteration details like copying the schema, and pulling down the weights to compute distances is done.
- A moving average is maintained over the past 25 weights. Iteration stops when the new weight is less than EPS away from the average.

Aside from that there's an interesting -bug?- that comes up as a result of this. Notice on line 70 how the new variables are bound each iteration? Well the underlying PigContext object just keeps adding these new variables instead of overwriting them. What this means is that, after a couple thousand iterations, depending on your PIG_HEAPSIZE env variable, the driver script will crash from an out of memory error. Yikes.

run it

Finally we get to run it. That part's easy:

```
$: export PIG_HEAPSIZE=4000
$: pig driver.py fit_line.pig data data.tsv 2
```

That is, we use the pig command to launch the driver program. The arguments to the driver script itself follow. We're running the fit_line.pig script where the data dir (where intermediate weights will go) exists under 'data' and the input data, data.tsv, should exist there as well. The '2' indicates we've got two weights (w0, and w1). The pig heapsize is to deal with the bug mentioned in the previous section.

On my laptop, in local mode, the convergence criteria was met after 1527 iterations.

results

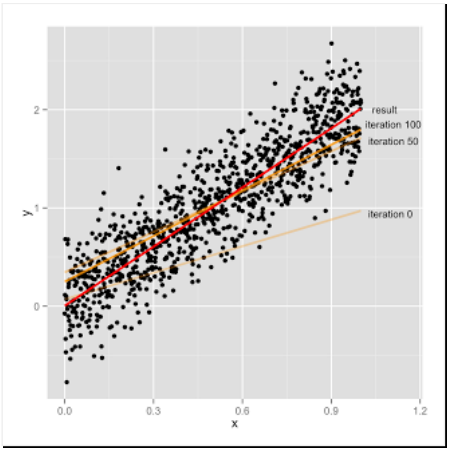
After 1527 iterations the weights ended up as (0.0040666759502969215, 2.0068029003414014) which is exactly what we'd expect. In other words:

$$y = 0.0041 + 2.0068x$$

which is the 'best fit' line to our original:

$$y = 0.3r(x) + 2x$$

And here's an illustration of what happened.



Looks pretty reasonable to me. Hurray. Now go fit some real data.

regression, python driver

2 comments:

rjurney November 14, 2013 at 7:28 PM

Awesomiddityawesome!

Reply

han yufei June 18, 2014 at 12:52 AM

I've tried it on the cluster but it doesn't work :(When it reaches 'zipped = Zip($w.weights, vector);', I get the runtime exception error 'ERROR 0: Scalar has more than one row in the output. That is very confused. weight.weights is just one single tuple, like vector, why can't we use it as input argument of udf ?

'

Reply