Regression Week 2: Multiple Regression (gradient descent)

In the first notebook we explored multiple regression using graphlab create. Now we will use graphlab along with numpy to solve for the regression weights with gradient descent.

In this notebook we will cover estimating multiple regression weights via gradient descent. You will:

- Add a constant column of 1's to a graphlab SFrame to account for the intercept
- Convert an SFrame into a Numpy array
- Write a predict_output() function using Numpy
- Write a numpy function to compute the derivative of the regression weights with respect to a single feature
- Write gradient descent function to compute the regression weights given an initial weight vector, step size and tolerance.
- Use the gradient descent function to estimate regression weights for multiple features

Fire up graphlab create

Make sure you have the latest version of graphlab (>= 1.7)

In [1]:

```
import graphlab
```

Load in house sales data

Dataset is from house sales in King County, the region where the city of Seattle, WA is located.

In [2]:

```
sales = graphlab.SFrame('C:/courses/Coursera/Current/ML Regression/Week2/kc_house_dat
```

```
[INFO] 1449552658 : INFO:     (initialize_globals_from_environment:28
2): Setting configuration variable GRAPHLAB_FILEIO_ALTERNATIVE_SSL_CER
T_FILE to C:\Users\Sandipan.Dey\AppData\Local\Dato\Dato Launcher\lib\si
te-packages\certifi\cacert.pem
1449552658 : INFO:     (initialize_globals_from_environment:282): Setti
ng configuration variable GRAPHLAB_FILEIO_ALTERNATIVE_SSL_CERT_DIR to
This non-commercial license of GraphLab Create is assigned to sandipa
n.dey@gmail.com and will expire on October 12, 2016. For commercial lic
ensing options, visit https://dato.com/buy/. (https://dato.com/buy/.)

[INFO] Start server at: ipc:///tmp/graphlab_server-9816 - Server binar
y: C:\Users\Sandipan.Dey\AppData\Local\Dato\Dato Launcher\lib\site-pack
ages\graphlab\unity_server.exe - Server log: C:\Users\Sandipan.Dey\AppD
ata\Local\Temp\graphlab_server_1449552658.log.0
[INFO] GraphLab Server Version: 1.7.1
```

If we want to do any "feature engineering" like creating new features or adjusting existing ones we should do this directly using the SFrames as seen in the other Week 2 notebook. For this notebook, however, we will work with the existing features.

Convert to Numpy Array

Although SFrames offer a number of benefits to users (especially when using Big Data and built-in graphlab functions) in order to understand the details of the implementation of algorithms it's important to work with a library that allows for direct (and optimized) matrix operations. Numpy is a Python solution to work with matrices (or any multi-dimensional "array").

Recall that the predicted value given the weights and the features is just the dot product between the feature and weight vector. Similarly, if we put all of the features row-by-row in a matrix then the predicted value for *all* the observations can be computed by right multiplying the "feature matrix" by the "weight vector".

First we need to take the SFrame of our data and convert it into a 2D numpy array (also called a matrix). To do this we use graphlab's built in .to_dataframe() which converts the SFrame into a Pandas (another python library) dataframe. We can then use Panda's .as_matrix() to convert the dataframe into a numpy matrix.

In [3]:

```
import numpy as np # note this allows us to refer to numpy as np instead
```

Now we will write a function that will accept an SFrame, a list of feature names (e.g. ['sqft_living', 'bedrooms']) and an target feature e.g. ('price') and will return two things:

- A numpy matrix whose columns are the desired features plus a constant column (this is how we create an 'intercept')

- A numpy array containing the values of the output

With this in mind, complete the following function (where there's an empty line you should write a line of code that does what the comment above indicates)

Please note you will need GraphLab Create version at least 1.7.1 in order for .to_numpy() to work!

In [4]:

```
def get_numpy_data(data_sframe, features, output):
    data_sframe['constant'] = 1 # this is how you add a constant column to an SFrame
    # add the column 'constant' to the front of the features list so that we can extr
    features = ['constant'] + features # this is how you combine two lists
    # select the columns of data_SFrame given by the features list into the SFrame fe
    features_sframe = data_sframe[features]
    print type(features_sframe)
    # the following line will convert the features_SFrame into a numpy matrix:
    feature_matrix = features_sframe.to_numpy()
    # assign the column of data_sframe associated with the output to the SArray outpu
    output_sarray = data_sframe[output]
    # the following will convert the SArray into a numpy array by first converting it
    output_array = output_sarray.to_numpy()
    return(feature_matrix, output_array)
```

For testing let's use the 'sqft_living' feature and a constant as our features and price as our output:

In [5]:

```
(example_features, example_output) = get_numpy_data(sales, ['sqft_living'], 'price')
print example_features[0,:] # this accesses the first row of the data the ':' indicat
print example_output[0] # and the corresponding output
```

```
<class 'graphlab.data_structures.sframe.SFrame'>
[  1.00000000e+00   1.18000000e+03]
221900.0
```

Predicting output given regression weights

Suppose we had the weights [1.0, 1.0] and the features [1.0, 1180.0] and we wanted to compute the predicted output 1.0*1.0 + 1.0*1180.0 = 1181.0 this is the dot product between these two arrays. If they're numpy arrayws we can use np.dot() to compute this:

In [6]:

```
my_weights = np.array([1., 1.]) # the example weights
my_features = example_features[0,] # we'll use the first data point
predicted_value = np.dot(my_features, my_weights)
print predicted_value
```

```
1181.0
```

np.dot() also works when dealing with a matrix and a vector. Recall that the predictions from all the observations is just the RIGHT (as in weights on the right) dot product between the features *matrix* and the weights *vector*. With this in mind finish the following predict_output function to compute the predictions for an entire matrix of features given the matrix and the weights:

In [7]:

```
def predict_output(feature_matrix, weights):
    # assume feature_matrix is a numpy matrix containing the features as columns and
    # create the predictions vector by using np.dot()
    predictions = np.dot(feature_matrix, weights)
    return(predictions)
```

If you want to test your code run the following cell:

In [8]:

```
test_predictions = predict_output(example_features, my_weights)
print test_predictions[0] # should be 1181.0
print test_predictions[1] # should be 2571.0
```

```
1181.0
2571.0
```

Computing the Derivative

We are now going to move to computing the derivative of the regression cost function. Recall that the cost function is the sum over the data points of the squared difference between an observed output and a predicted output.

Since the derivative of a sum is the sum of the derivatives we can compute the derivative for a single data point and then sum over data points. We can write the squared difference between the observed output and predicted output for a single point as follows:

(w[0]*[CONSTANT] + w[1]*[feature_1] + ... + w[i] *[feature_i] + ... + w[1]*[feature_k] - output)^2

Where we have k features and a constant. So the derivative with respect to weight w[i] by the chain rule is:

2*(w[0]*[CONSTANT] + w[1]*[feature_1] + ... + w[i] *[feature_i] + ... + w[1]*[feature_k] - output)*
[feature_i]

The term inside the paranethesis is just the error (difference between prediction and output). So we can re-write this as:

2*error*[feature_i]

That is, the derivative for the weight for feature i is the sum (over data points) of 2 times the product of the error and the feature itself. In the case of the constant then this is just twice the sum of the errors!

Recall that twice the sum of the product of two vectors is just twice the dot product of the two vectors. Therefore the derivative for the weight for feature_i is just two times the dot product between the values of feature_i and the current errors.

With this in mind complete the following derivative function which computes the derivative of the weight given the value of the feature (over all data points) and the errors (over all data points).

In [11]:

```
def feature_derivative(errors, feature):
    # Assume that errors and feature are both numpy arrays of the same length (number
    # compute twice the dot product of these vectors as 'derivative' and return the v
    derivative = 2*np.dot(errors, feature)
    return(derivative)
```

To test your feature derivartive run the following:

In [12]:

```
(example_features, example_output) = get_numpy_data(sales, ['sqft_living'], 'price')
my_weights = np.array([0., 0.]) # this makes all the predictions 0
test_predictions = predict_output(example_features, my_weights)
# just like SFrames 2 numpy arrays can be elementwise subtracted with '-':
errors = test_predictions - example_output # prediction errors in this case is just t
feature = example_features[:,0] # let's compute the derivative with respect to 'const
derivative = feature_derivative(errors, feature)
print derivative
print -np.sum(example_output)*2 # should be the same as derivative
```

```
<class 'graphlab.data_structures.sframe.SFrame'>
-23345850022.0
-23345850022.0
```

Gradient Descent

Now we will write a function that performs a gradient descent. The basic premise is simple. Given a starting point we update the current weights by moving in the negative gradient direction. Recall that the gradient is the direction of *increase* and therefore the negative gradient is the direction of *decrease* and we're trying to *minimize* a cost function.

The amount by which we move in the negative gradient *direction* is called the 'step size'. We stop when we are 'sufficiently close' to the optimum. We define this by requiring that the magnitude (length) of the gradient vector to be smaller than a fixed 'tolerance'.

With this in mind, complete the following gradient descent function below using your derivative function above. For each step in the gradient descent we update the weight for each feature befofe computing our stopping criteria

In [19]:

```
from math import sqrt # recall that the magnitude/length of a vector [g[0], g[1], g[2
```

In [20]:

```
def regression_gradient_descent(feature_matrix, output, initial_weights, step_size, t
    converged = False
    weights = np.array(initial_weights) # make sure it's a numpy array
    while not converged:
        # compute the predictions based on feature_matrix and weights using your pred
        predictions = predict_output(feature_matrix, weights)
        # compute the errors as predictions - output
        errors = predictions - output
        gradient_sum_squares = 0 # initialize the gradient sum of squares
        # while we haven't reached the tolerance yet, update each feature's weight
        for i in range(len(weights)): # loop over each weight
            # Recall that feature_matrix[:, i] is the feature column associated with
            # compute the derivative for weight[i]:
            derivative = feature_derivative(errors, feature_matrix[:, i])
            # add the squared value of the derivative to the gradient magnitude (for
            gradient_sum_squares += derivative**2
            # subtract the step size times the derivative from the current weight
            weights[i] -= step_size * derivative
        # compute the square-root of the gradient sum of squares to get the gradient
        gradient_magnitude = sqrt(gradient_sum_squares)
        if gradient_magnitude < tolerance:
            converged = True
    return(weights)
```

A few things to note before we run the gradient descent. Since the gradient is a sum over all the data points and involves a product of an error and a feature the gradient itself will be very large since the features are large (squarefeet) and the output is large (prices). So while you might expect "tolerance" to be small, small is only relative to the size of the features.

For similar reasons the step size will be much smaller than you might expect but this is because the gradient has such large values.

Running the Gradient Descent as Simple Regression

First let's split the data into training and test data.

In [21]:

```
train_data,test_data = sales.random_split(.8,seed=0)
```

Although the gradient descent is designed for multiple regression since the constant is now a feature we can use the gradient descent function to estimat the parameters in the simple regression on squarefeet. The folowing cell sets up the feature_matrix, output, initial weights and step size for the first model:

In [22]:

```
# let's test out the gradient descent
simple_features = ['sqft_living']
my_output = 'price'
(simple_feature_matrix, output) = get_numpy_data(train_data, simple_features, my_outp
initial_weights = np.array([-47000., 1.])
step_size = 7e-12
tolerance = 2.5e7
```

```
<class 'graphlab.data_structures.sframe.SFrame'>
```

Next run your gradient descent with the above parameters.

In [26]:

```
weights = regression_gradient_descent(simple_feature_matrix, output, initial_weights,
weights
```

Out[26]:

```
array([-46999.88716555,    281.91211912])
```

How do your weights compare to those achieved in week 1 (don't expect them to be exactly the same)?

Quiz Question: What is the value of the weight for sqft_living -- the second element of ‘simple_weights’ (rounded to 1 decimal place)?

Use your newly estimated weights and your predict_output() function to compute the predictions on all the TEST data (you will need to create a numpy array of the test feature_matrix and test output first:

In [27]:

```
(test_simple_feature_matrix, test_output) = get_numpy_data(test_data, simple_features
```

```
<class 'graphlab.data_structures.sframe.SFrame'>
```

Now compute your predictions using test_simple_feature_matrix and your weights from above.

In [29]:

```
predictions = predict_output(test_simple_feature_matrix, weights)
```

Quiz Question: What is the predicted price for the 1st house in the TEST data set for model 1 (round to nearest dollar)?

In [30]:

```
predictions[0]
```

Out[30]:

```
356134.44317092974
```

Now that you have the predictions on test data, compute the RSS on the test data set. Save this value for comparison later. Recall that RSS is the sum of the squared errors (difference between prediction and output).

In [31]:

```
RSS = sum((predictions - test_output)**2)
RSS
```

Out[31]:

```
275400047593155.69
```

Running a multiple regression

Now we will use more than one actual feature. Use the following code to produce the weights for a second model with the following parameters:

In [34]:

```
model_features = ['sqft_living', 'sqft_living15'] # sqft_living15 is the average squa
my_output = 'price'
(feature_matrix, output) = get_numpy_data(train_data, model_features, my_output)
initial_weights = np.array([-100000., 1., 1.])
step_size = 4e-12
tolerance = 1e9
```

```
<class 'graphlab.data_structures.sframe.SFrame'>
```

Use the above parameters to estimate the model weights. Record these values for your quiz.

In [35]:

```
weights = regression_gradient_descent(feature_matrix, output, initial_weights, step_s
weights
```

Out[35]:

```
array([ -9.99999688e+04,   2.45072603e+02,   6.52795277e+01])
```

Use your newly estimated weights and the predict_output function to compute the predictions on the TEST data. Don't forget to create a numpy array for these features from the test set first!

In [38]:

```
(test_feature_matrix, test_output) = get_numpy_data(test_data, model_features, my_out
predictions = predict_output(test_feature_matrix, weights)
```

```
<class 'graphlab.data_structures.sframe.SFrame'>
```

Quiz Question: What is the predicted price for the 1st house in the TEST data set for model 2 (round to nearest dollar)?

In [39]:

```
predictions[0]
```

Out[39]:

```
366651.41203655908
```

What is the actual price for the 1st house in the test data set?

In [40]:

```
test_output[0]
```

Out[40]:

```
310000.0
```

Quiz Question: Which estimate was closer to the true price for the 1st house on the Test data set, model 1 or model 2?

Now use your predictions and the output to compute the RSS for model 2 on TEST data.

In [ ]:

```

```

Quiz Question: Which model (1 or 2) has lowest RSS on all of the TEST data?

In [41]:

```
RSS = sum((predictions - test_output)**2)
RSS
```

Out[41]:

```
270263446465243.97
```

In [ ]:

```

```