# Regression Week 2: Multiple Linear Regression Quiz 2

Estimating Multiple Regression Coefficients (Gradient Descent)

In the first notebook we explored multiple regression using GraphLab Create. Now we will use SFrames 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 SFrame (or otherwise) 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

## If you are doing the assignment with IPython Notebook

An IPython Notebook has been provided below to you for this quiz. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

## What you need to download

## If you are using GraphLab Create:

- Download the King County House Sales data In SFrame format: kc_house_data.gl.zip
- Download the companion IPython Notebook: week-2-multiple-regression-assignment-2-blank.ipynb
- Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

**Upgrade GraphLab Create.** If you are using GraphLab Create and already have it installed, please make sure you upgrade to the latest version before doing this assignment. The simplest way to do this is to:

1. Open the Dato Launcher.
2. Click on 'TERMINAL'.
3. On the terminal window, type:

```
pip install --upgrade graphlab-create
```

## If you are not using GraphLab Create:

- Download the King County House Sales data csv file: kc_house_data.csv
- Download the King County House Sales training data csv file: kc_house_train_data.csv
- Download the King County House Sales testing data csv file: kc_house_test_data.csv
- **IMPORTANT: use the following types for columns when importing the csv files. Otherwise, they may not be imported correctly: [str, str, float, float, float, float, int, str, int, int, int, int, int, int, int, int, str, float, float, float, float]. If your tool of choice requires a dictionary of types for importing csv files (e.g. Pandas), use:**

```
dtype_dict = {'bathrooms':float, 'waterfront':int, 'sqft_above':int, 'sqft_living15':
float, 'grade':int, 'yr_renovated':int, 'price':float, 'bedrooms':float, 'zipcode':st
r, 'long':float, 'sqft_lot15':float, 'sqft_living':float, 'floors':str, 'condition':i
nt, 'lat':float, 'date':str, 'sqft_basement':int, 'yr_built':int, 'id':str, 'sqft_lot
':int, 'view':int}
```

## Useful resources

You may need to install the software tools or use the free Amazon EC2 machine. Instructions for both options are provided in the reading for Module 1.

If you are following the IPython Notebook and/or are new to numpy then you might find the following tutorial helpful: numpy-tutorial.ipynb

## If instead you are using other tools to do your homework

You are welcome, however, to write your own code and use any other libraries, like Pandas or R, to help you in the process. If you would like to take this path, follow the instructions below.

**1.** If you’re using SFrames, import graphlab and load in the house data (this is the graphlab command you can also download the csv). e.g. in python with SFrames:

```
sales = graphlab.SFrame('kc_house_data.gl/')
```

**2.** If you’re using python: to do the matrix operations required to perform a gradient descent we will be using the popular python library ‘numpy’ which is a computational library specialized for operations on arrays. For students unfamiliar with numpy we have created a numpy tutorial (see useful resources). It is common to import numpy under the name ‘np’ for short, to do this execute:

```
import numpy as np
```

**3.** Next write a function that takes a data set, a list of features (e.g. [‘sqft_living’, ‘bedrooms’]), to be used as inputs, and a name of the output (e.g. ‘price’). This function should return a features_matrix (2D array) consisting of first a column of ones followed by columns containing the values of the input features in the data set in the same order as the input list. It should also return an output_array which is an array of the values of the output in the data set (e.g. ‘price’). e.g. if you’re using SFrames and numpy you can complete the following function:

```
def get_numpy_data(data_sframe, features, output):
    data_sframe['constant'] = 1 # add a constant column to an SFrame
    # prepend variable 'constant' to the features list
    features = ['constant'] + features
    # select the columns of data_SFrame given by the ‘features’ list into the SFrame
‘features_sframe’


    # this will convert the features_sframe into a numpy matrix with GraphLab Create
>= 1.7!!
    features_matrix = features_sframe.to_numpy()
    # assign the column of data_sframe associated with the target to the variable ‘ou
tput_sarray’


    # this will convert the SArray into a numpy array:
    output_array = output_sarray.to_numpy() # GraphLab Create>= 1.7!!
    return(features_matrix, output_array)
```

In order for the .to_numpy() command to work ensure that you have GraphLab Create version 1.7 or greater.

**4.** If the features matrix (including a column of 1s for the constant) is stored as a 2D array (or matrix) and the regression weights are stored as a 1D array then the predicted output is just the dot product between the features matrix and the weights (with the weights on the right). Write a function ‘predict_output’ which accepts a 2D array ‘feature_matrix’ and a 1D array ‘weights’ and returns a 1D array ‘predictions’. e.g. in python:

```
def predict_outcome(feature_matrix, weights):
    [your code here]
    return(predictions)
```

**5.** If we have a the values of a single input feature in an array ‘feature’ and the prediction ‘errors’ (predictions - output) then the derivative of the regression cost function with respect to the weight of ‘feature’ is just twice the dot product between ‘feature’ and ‘errors’. Write a function that accepts a ‘feature’ array and ‘error’ array and returns the ‘derivative’ (a single number). e.g. in python:

```
def feature_derivative(errors, feature):
    [your code here]
    return(derivative)
```

**6.** Now we will use our predict_output and feature_derivative to write a gradient descent function. Although we can compute the derivative for all the features simultaneously (the gradient) we will explicitly loop over the features individually for simplicity. Write a gradient descent function that does the following:

- Accepts a numpy feature_matrix 2D array, a 1D output array, an array of initial weights, a step size and a convergence tolerance.
- While not converged updates each feature weight by subtracting the step size times the derivative for that feature given the current weights
- At each step computes the magnitude/length of the gradient (square root of the sum of squared components)
- When the magnitude of the gradient is smaller than the input tolerance returns the final weight vector.

e.g. if you’re using SFrames and numpy you can complete the following function:

```
def regression_gradient_descent(feature_matrix, output, initial_weights, step_size, tolerance):
    converged = False
    weights = np.array(initial_weights)
    while not converged:
        # compute the predictions based on feature_matrix and weights:
        # compute the errors as predictions - output:

        gradient_sum_squares = 0 # initialize the gradient
        # while not converged, update each weight individually:
        for i in range(len(weights)):
            # Recall that feature_matrix[:, i] is the feature column associated with weights[i]
            # compute the derivative for weight[i]:

            # add the squared derivative to the gradient magnitude

            # update the weight based on step size and derivative:

        gradient_magnitude = sqrt(gradient_sum_squares)
        if gradient_magnitude < tolerance:
            converged = True
    return(weights)
```

**7.** Now split the sales data into training and test data. Like previous notebooks it’s important to use the same seed.

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

For those students not using SFrames please download the training and testing data csv files.

**8.** Now we will run the regression_gradient_descent function on some actual data. In particular we will use the gradient descent to estimate the model from Week 1 using just an intercept and slope. Use the following parameters:

- features: ‘sqft_living’

- output: ‘price’
- initial weights: -47000, 1 (intercept, sqft_living respectively)
- step_size = 7e-12
- tolerance = 2.5e7

e.g. in python with numpy and SFrames:

```
simple_features = ['sqft_living']

my_output= 'price'

(simple_feature_matrix, output) = get_numpy_data(train_data, simple_features, my_outp
ut)

initial_weights = np.array([-47000., 1.])

step_size = 7e-12

tolerance = 2.5e7
```

Use these parameters to estimate the slope and intercept for predicting prices based only on ‘sqft_living’.

e.g. using python:

```
simple_weights = regression_gradient_descent(simple_feature_matrix, output,initial_we
ights, step_size,                                              tolerance)
```

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

**10.** Now build a corresponding ‘test_simple_feature_matrix’ and ‘test_output’ using test_data. Using ‘test_simple_feature_matrix’ and ‘simple_weights’ compute the predicted house prices on all the test data.

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

**12.** Now compute RSS on all test data for this model. Record the value and store it for later

**13.** Now we will use the gradient descent to fit a model with more than 1 predictor variable (and an intercept). Use the following parameters:

- model features = ‘sqft_living’, ‘sqft_living_15’
- output = ‘price’

- initial weights = [-100000, 1, 1] (intercept, sqft_living, and sqft_living_15 respectively)
- step size = 4e-12
- tolerance = 1e9

e.g. in python with numpy and SFrames:

```
model_features = ['sqft_living', 'sqft_living15']
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
```

*Note that sqft_living_15 is the average square feet of the nearest 15 neighbouring houses.*

Run gradient descent on a model with ‘sqft_living’ and ‘sqft_living_15’ as well as an intercept with the above parameters. Save the resulting regression weights.

**14.** Use the regression weights from this second model (using sqft_living and sqft_living_15) and predict the outcome of all the house prices on the TEST data.

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

**16.** What is the actual price for the 1st house in the Test data set?

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

**18.** Now compute RSS on all test data for the second model. Record the value and store it for later.

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