ICS 485: Machine Learning

Programming Assignment: **Extra-2**

[Ridge Regression Assignment]

Part I

In this assignment, we will run ridge regression multiple times with different L_2-norm penalties to see which one produces the best fit. We will revisit the example of polynomial regression as a means to see the effect of L_2-norm regularization. In particular, we will:

1- Use a pre-built implementation of regression to run polynomial regression.
2- Use **matplotlib** to visualize polynomial regressions.
3- Use a pre-built implementation of regression to run polynomial regression, this time with L_2-norm penalty
4- Use matplotlib to visualize polynomial regressions under L_2-norm regularization
5- Choose best L_2-norm penalty using cross-validation.
6- Assess the final fit using test data.

We will continue to use the House data from previous assignment (Ext-1).

Choice of tools

1- For the purpose of this assessment, you may choose between GraphLab Create and scikit-learn (with Pandas).

2- If you are using GraphLab Create, download the IPython notebook and follow the instructions contained in the notebook.

3- If you are using Pandas+scikit-learn combination, follow through the instructions in this reading.

What you need to get

A- If you are using GraphLab Create:
   a. King County House Sales data In SFrame format: kc_house_data.gl.zip
   b. The companion IPython Notebook.
   c. Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

B- If you are not using GraphLab Create:
   a. King County House Sales data csv file: **kc_house_data.csv.zip**
   b. King County House Sales training data csv file: **kc_house_train_data.csv.zip**
   c. King County House Sales validation data csv file: **kc_house_valid_data.csv.zip**
   d. King County House Sales testing data csv file: **kc_house_test_data.csv.zip**
   e. csv file containing randomly shuffled rows of training and validation data combined: **kc_house_train_valid_shuffled.csv.zip**
   f. King County House Sales subset 1 data csv file: **kc_house_set_1_data.csv.zip**
   g. King County House Sales subset 2 data csv file: **kc_house_set_2_data.csv.zip**
   h. King County House Sales subset 3 data csv file: **kc_house_set_3_data.csv.zip**
   i. King County House Sales subset 4 data csv file: **kc_house_set_4_data.csv.zip**

All the required data files and Python notebook are available in the **Assignment-Ext2_PartA.rar** file

Note:
1- If you are using GraphLab Create and the companion IPython Notebook:
   a. Open the companion IPython notebook and follow the instructions in the notebook.

2- If you are using scikit-learn with Pandas
   a. Use the attached data files

For the remainder of the assignment we will be working with the house Sales data. Load in the data and also sort the sales data frame by 'sqft_living'. When we plot the fitted values we want to join them up in a line and this works best if the variable on the X-axis (which will be 'sqft_living') is sorted. For houses with identical square footage, we break the tie by their prices.

Task 1: Let us consider a 15th-order polynomial model using the 'sqft_living' input. Generate polynomial features up to degree 15 using **`polynomial_sframe()`** and fit a model with these features. When fitting the model, use an L_2-norm penalty of $\mathbf{1.5e^{-5}}$:

l2_small_penalty = 1.5e-5

Note: When we have so many features and so few data points, the solution can become highly numerically unstable, which can sometimes lead to strange unpredictable results. Thus, rather than using no regularization, we will introduce a tiny amount of regularization (**l2_penalty=$1.5e^{-5}$**) to make the solution numerically stable.

With the L_2-norm penalty specified above, fit the model and print out the learned weights. Add "**alpha=l2_small_penalty**" and "**normalize=True**" to the parameter list of linear_model.Ridge:

```
from sklearn import linear_model
import numpy as np
poly15_data = polynomial_sframe(sales['sqft_living'], 15) # use equivalent of
  `polynomial_sframe`
model = linear_model.Ridge(alpha=l2_small_penalty, normalize=True)
model.fit(poly15_data, sales['price'])
```

Analysis Question 1: What is the learned value for the coefficient of feature **power_1**?

Task 2: When we split the sales data into four subsets and fit the model of degree 15, the result came out to be very different for each subset. The model had a high variance. We will see in a moment that ridge regression reduces such variance.

For this task, please download the provided csv files for each subset and load them with the given list of types:

```
set_1 = pd.read_csv('wk3_kc_house_set_1_data.csv', dtype=dtype_dict)
set_2 = pd.read_csv('wk3_kc_house_set_2_data.csv', dtype=dtype_dict)
set_3 = pd.read_csv('wk3_kc_house_set_3_data.csv', dtype=dtype_dict)
set_4 = pd.read_csv('wk3_kc_house_set_4_data.csv', dtype=dtype_dict)
```

Fit a 15th degree polynomial on each of the 4 sets, plot the results and view the weights for the four models. This time, set:

l2_small_penalty=1e-9

and use this value for the L_2-norm penalty. Make sure to add "**alpha=l2_small_penalty**" and "**normalize=True**" to the parameter list of linear_model.Ridge.

The four curves should differ from one another a lot, as should the coefficients you learned.

Analysis Question 2: For the models learned in each of these training sets, what are the smallest and largest values you learned for the coefficient of feature **power_1**?
Note: For the purpose of answering this question, negative numbers are considered "smaller" than positive numbers. So -5 is smaller than -3, and -3 is smaller than 5 and so forth.)

Task 3: Generally, whenever we see weights change so much in response to change in data, we believe the variance of our estimate to be large. Ridge regression aims to address this issue by penalizing "large" weights.
Note: The weights looked quite small, but they are not that small because 'sqft_living' input is in the order of thousands.

Fit a 15th-order polynomial model on set_1, set_2, set_3, and set_4, this time with a large L_2-norm penalty. Make sure to add "**alpha=l2_large_penalty**" and "**normalize=True**" to the parameter list, where the value of **l2_large_penalty** is given by:

l2_large_penalty=1.23e2

These curves should vary a lot less, now that you introduced regularization.

Analysis Question 3: For the models learned with regularization in each of these training sets, what are the smallest and largest values you learned for the coefficient of feature power_1?
Note: For the purpose of answering this question, negative numbers are considered "smaller" than positive numbers. So -5 is smaller than -3, and -3 is smaller than 5 and so forth.

Task 4: Just like the polynomial degree, the L_2-norm penalty is a "magic" parameter we need to select. We could use the validation set approach as we did in the last module, but that approach has a major disadvantage: it leaves fewer observations available for training. Cross-validation seeks to overcome this issue by using all of the training set in a smart way.

We will implement a kind of cross-validation called k-fold cross-validation. The method gets its name because it involves dividing the training set into k segments of roughly equal size. Similar to the validation set method, we measure the validation error with one of the segments designated as the validation set. The major difference is that we repeat the process k times as follows:

a. Set aside segment 0 as the validation set, and fit a model on rest of data, and evaluate it on this validation set
b. Set aside segment 1 as the validation set, and fit a model on rest of data, and evaluate it on this validation set
c. Carry on with other segments ...
d. Set aside segment k-1 as the validation set, and fit a model on rest of data, and evaluate it on this validation set
e. After this process, we compute the average of the k validation errors, and use it as an estimate of the generalization error. Notice that all observations are used for both training and validation, as we iterate over segments of data.

To estimate the generalization error well, it is crucial to shuffle the training data before dividing them into segments. We reserve 10% of the data as the test set and randomly shuffle the remainder. Let us call the shuffled data '**train_valid_shuffled**'.

For the purpose of this assignment, let us download the csv file containing pre-shuffled rows of training and validation sets combined: **kc_house_train_valid_shuffled.csv**. In practice, you would shuffle the rows with a dynamically determined random seed.
train_valid_shuffled = pd.read_csv('kc_house_train_valid_shuffled.csv', dtype=dtype_dict)
test = pd.read_csv('kc_house_test_data.csv', dtype=dtype_dict)

Divide the combined training and validation set into equal segments. Each segment should receive n/k elements, where n is the number of observations in the training set and k is the number of

segments. Since the segment 0 starts at index 0 and contains n/k elements, it ends at index (n/k)-1. The segment 1 starts where the segment 0 left off, at index (n/k). With n/k elements, the segment 1 ends at index (n*2/k)-1. Continuing in this fashion, we deduce that the segment i starts at index (n*i/k) and ends at (n*(i+1)/k)-1.

With this pattern in mind, we write a short loop that prints the starting and ending indices of each segment, just to make sure you are getting the splits right.

```
n = len(train_valid_shuffled)
k = 10 # 10-fold cross-validation
for i in xrange(k):
    start = (n*i)/k
    end = (n*(i+1))/k-1
    print i, (start, end)
```

Let us familiarize ourselves with array slicing with Pandas. To extract a continuous slice from a DataFrame, use colon in square brackets. For instance, the following cell extracts rows 0 to 9 of train_valid_shuffled. Notice that the first index (0) is included in the slice but the last index (10) is omitted.

```
train_valid_shuffled[0:10] # select rows 0 to 9
```

If the observations are grouped into 10 segments, the segment i is given by

```
start = (n*i)/10
end = (n*(i+1))/10
train_valid_shuffled[start:end+1]
```

Meanwhile, to choose the remainder of the data that's not part of the segment i, we select two slices (0: start) and (end+1: n) and paste them together.

```
train_valid_shuffled[0:start].append(train_valid_shuffled[end+1:n])
```

Now we are ready to implement k-fold cross-validation. Write a function that computes k validation errors by designating each of the k segments as the validation set. It accepts as parameters (i) k, (ii) l2_penalty, (iii) dataframe containing input features (e.g. poly15_data) and (iv) column of output values (e.g. price). The function returns the average validation error using k segments as validation sets. We shall assume that the input dataframe does not contain the output column.

For each i in [0, 1, ... k-1]:

i. Compute starting and ending indices of segment i and call 'start' and 'end'
ii. Form validation set by taking a slice (start:end+1) from the data.
iii. Form training set by appending slice (end+1:n) to the end of slice (0:start).
iv. Train a linear model using training set just formed, with a given l2_penalty
v. Compute validation error (RSS) using validation set just formed

For example, in Python:

def k_fold_cross_validation(k, l2_penalty, data, output):

...

return [average validation error]

Task 5: Once we have a function to compute the average validation error for a model, we can write a loop to find the model that minimizes the average validation error. Write a loop that does the following:

i. We will again be aiming to fit a 15th-order polynomial model using the sqft_living input
ii. For each l2_penalty in [10^3, $10^{3.5}$, 10^4, $10^{4.5}$, ..., 10^9] (to get this in Python, you can use this Numpy function: **np.logspace(3, 9, num=13)**.): Run 10-fold cross-validation with l2_penalty.
iii. Report which L_2-norm penalty produced the lowest average validation error.

Note: Since the degree of the polynomial is now fixed to 15, to make things faster, you should generate polynomial features in advance and re-use them throughout the loop. Make sure to use train_valid_shuffled when generating polynomial features!

Analysis Question 4: What is the best value for the L_2-norm penalty according to 10-fold validation?

Note: Once you found the best value for the L_2-norm penalty using cross-validation, it is important to retrain a final model on all of the training data using this value of l2_penalty. This way, your final model will be trained on the entire dataset.

Analysis Question 5: Using the best L_2-norm penalty found above, train a model using all training data. What is the RSS on the TEST data of the model you learn with this L_2-norm penalty?

PART II

In this assignment, you will implement ridge regression via gradient descent. You will:

1. Convert an SFrame into a Numpy array (if applicable)
2. Write a Numpy function to compute the derivative of the regression weights with respect to a single feature
3. Write gradient descent function to compute the regression weights given an initial weight vector, step size, tolerance, and L_2-norm penalty

What you need to get:

A- If you are using GraphLab Create:
   a. King County House Sales data In SFrame format: **kc_house_data.gl.zip**
   b. The companion IPython Notebook: **Assignmet-Ext2-partB.ipynb**
   c. Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

B- If you are not using GraphLab Create:
   a. King County House Sales data csv file: kc_house_data.csv
   b. King County House Sales data csv file: kc_house_train_data.csv
   c. King County House Sales data csv file: kc_house_test.csv

All the required data files and Python notebook are available in the **Assignment-Ext2_PartB.rar** file

Data Preparation: If you are using SFrame, import GraphLab Create and load in the house data as follows:

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

Otherwise, load all the three csv files.

Note:
If you are 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.

import numpy as np

Pre-Task 1: Write '**get_numpy_data**' function that takes a dataframe, 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 returns a 'feature_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 alsos return an 'output_array' which is an array of the values of the output in the data set (e.g. 'price').

Example in Python:

```
def get_numpy_data(data_sframe, features, output):
    ...
    return (feature_matrix, output_array)
```

Pre-Task 2: Write a function '**predict_output**' function. This function accepts a 2D array 'feature_matrix' and a 1D array 'weights' and return a 1D array 'predictions'.

Example in Python:

```
def predict_output(feature_matrix, weights):
    ...
    return predictions
```

Task 1: 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, plus the L_2-norm penalty term.

Cost(w) = SUM[(prediction - output)^2] + l2_penalty*(w[0]^2 + w[1]^2 + ... + w[k]^2)

Note: Important to note that this formulation is little different than what we saw in the class. Although both are valid formulations.

Since the derivative of a sum is the sum of the derivatives, we can take the derivative of the first part (the RSS) and add the derivative of the regularization part. As we saw, the derivative of the RSS with respect to w[i] can be written as:

2*SUM[error*[feature_i]]

The derivative of the regularization term with respect to w[i] is:

2*l2_penalty*w[i]

Summing both, we get

2*SUM[error*[feature_i]] + 2*l2_penalty*w[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, plus 2*l2_penalty*w[i].

Note: We will not regularize the constant. Thus, in the case of the constant, the derivative is just twice the sum of the errors (without the 2*l2_penalty*w[0] term).

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, plus 2*l2_penalty*w[i].

With this in mind write the 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). To decide when to we are dealing with the constant (so we don't regularize it) we added the extra parameter to the call 'feature_is_constant' which you should set to True when computing the derivative of the constant and False otherwise.

```
def feature_derivative_ridge(errors, feature, weight, l2_penalty,
  feature_is_constant):
    ...
    return derivative
```

Task 2: To test your feature derivative function, run the following:

```
(example_features, example_output) = get_numpy_data(sales, ['sqft_living'],
  'price')
my_weights = np.array([1., 10.])
test_predictions = predict_output(example_features, my_weights)
errors = test_predictions - example_output # prediction errors
# next two lines should print the same values
print feature_derivative_ridge(errors, example_features[:,1], my_weights[1], 1,
  False)
print np.sum(errors*example_features[:,1])*2+20.
print ''
# next two lines should print the same values
print feature_derivative_ridge(errors, example_features[:,0], my_weights[0], 1,
  True)
print np.sum(errors)*2.
```

Task 3: 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 will set a maximum number of iterations and take gradient steps until we reach this maximum number. If no maximum number is supplied, the maximum should be set 100 by default. (Use default parameter values in Python.)

With this in mind, write a gradient descent function using your derivative function above. For each step in the gradient descent, we update the weight for each feature before computing our stopping criteria.
The function will take the following parameters:

i. 2D feature matrix
ii. array of output values
iii. initial weights
iv. step size
v. L_2-norm penalty
vi. maximum number of iterations

To make your job easier, a code skeleton in Python is provided:

```
def ridge_regression_gradient_descent(feature_matrix, output, initial_weights,
  step_size, l2_penalty, max_iterations=100):
    weights = np.array(initial_weights) # make sure it's a numpy array
      #while not reached maximum number of iterations:
      # compute the predictions using your predict_output() function
      # compute the errors as predictions - output
      for i in xrange(len(weights)): # loop over each weight
        # Recall that feature_matrix[:,i] is the feature column associated
          with weights[i]
        # compute the derivative for weight[i].
        #(Remember: when i=0, you are computing the derivative of the
          constant!)
        # subtract the step size times the derivative from the current
          weight
          return weights
```

Task 4: The L_2-norm penalty gets its name because it causes weights to have small L2 norms than otherwise. Let us see how large weights get penalized. Let us consider a simple model with 1 feature.

features: 'sqft_living'
output: 'price'

4.1 Split the dataset into training set and test set. If you are using GraphLab Create, call

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

Otherwise, please download the csv files from the download section.

4.2 Convert the training set and test set using the 'get_numpy_data' function.e.g. in Python:

```
simple_features = ['sqft_living']
my_output = 'price'
(simple_feature_matrix, output) = get_numpy_data(train_data, simple_features,
  my_output)
(simple_test_feature_matrix, test_output) = get_numpy_data(test_data,
  simple_features, my_output)
```

4.3 First, let us consider no regularization. Set the L_2-norm penalty to **0.0** and run your ridge regression algorithm to learn the weights of the simple model (described above). Use the following parameters:

step_size = 1e-12
max_iterations = 1000
initial_weights = all zeros

Store the learned weights as:

simple_weights_0_penalty

We will use them later.

4.4 Next, let us consider high regularization. Set the L_2-norm penalty to **$1e^{11}$** and run your ridge regression to learn the weights of the simple model. Use the same parameters as above. Call your weights:

simple_weights_high_penalty

We will use them later.

4.5 If you have access to matplotlib, the following piece of code will plot the two learned models. (The blue line is for the model with no regularization and the red line is for the one with high regularization.)

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(simple_feature_matrix,output,'k.',
    simple_feature_matrix,predict_output(simple_feature_matrix,
      simple_weights_0_penalty),'b-',
    simple_feature_matrix,predict_output(simple_feature_matrix,
      simple_weights_high_penalty),'r-')
```

If you do not have access to matplotlib, look at each set of coefficients. If you were to plot 'sqft_living' vs the price, which of the two coefficients is the slope and which is the intercept?

Analysis Question 1: What is the value of the coefficient for sqft_living that you learned with no regularization, rounded to 1 decimal place? What about the one with high regularization?

Analysis Question 2: Comparing the lines you fit with the one with no regularization versus high regularization, which one is steeper?

Task 5: Compute the RSS on the TEST data for the following three sets of weights:

i. The initial weights (all zeros)
ii. The weights learned with no regularization
iii. The weights learned with high regularization

Analysis Question 3: What are the RSS on the test data for each of the set of weights above (initial, no regularization, high regularization)?

Task 6: Let us now consider a model with 2 features: ['sqft_living', 'sqft_living_15']. First, create Numpy version of your training and test data with the two features.

Example in Python:
model_features = ['sqft_living', 'sqft_living15']
my_output = 'price'
(feature_matrix, output) = get_numpy_data(train_data, model_features, my_output)
(test_feature_matrix, test_output) = get_numpy_data(test_data, model_features, my_output)

6.1 First, let us consider no regularization. Set the L_2-norm penalty to **0.0** and run your ridge regression algorithm. Use the following parameters:

i. initial_weights = all zeros
ii. step size = 1e-12
iii. max_iterations = 1000

Call the learned weights:

multiple_weights_0_penalty

6.2 Next, let us consider high regularization. Set the L_2-norm penalty to $\mathbf{1e^{11}}$ and run your ridge regression to learn the weights of the simple model. Use the same parameters as above. Call your weights:

multiple_weights_high_penalty

Analysis Question 4: What is the value of the coefficient for 'sqft_living' that you learned with no regularization, rounded to 1 decimal place? What about the one with high regularization?

Task 7: Compute the RSS on the TEST data for the following three sets of weights:

i. The initial weights (all zeros)
ii. The weights learned with no regularization
iii. The weights learned with high regularization

Analysis Question 5: What are the RSS on the test data for each of the set of weights above (initial, no regularization, high regularization)?

Task 7: Predict the house price for the 1st house in the test set using the no regularization and high regularization models. (Remember that python starts indexing from 0.)

Analysis Question 6: What is the error in predicting the price of the first house in the test set using the weights learned with no regularization? What about with high regularization?