Lasso Regression with Coordinate Descent

In [21]:
```
%%javascript
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebook_toc.js')
```

Imports packages, data and functions

In [4]:
```
import graphlab
import numpy as np

# Dataset is from house sales in King County, the region where the city of Seattle, WA is locate
d.
sales = graphlab.SFrame('kc_house_data.gl/')
# floors' was defined as string, so convert to int
sales['floors'] = sales['floors'].astype(int)
```

In [5]:
```
# Put all features in matrix, output in array
def get_numpy_data(data_sframe, features, output):
    data_sframe['constant'] = 1
    features = ['constant'] + features
    features_sframe = data_sframe[features]
    feature_matrix = features_sframe.to_numpy()

    output_sarray = data_sframe[output]
    output_array = output_sarray.to_numpy()
    return(feature_matrix, output_array)
```

Normalize features

To give equal considerations for all features, we need to normalize features: we divide each feature by its 2-norm. **Normalization: per feature, divide datapoints by the root of the sum of the squares**

In [6]:
```
def normalize_features(feature_matrix):
    norms = np.linalg.norm(feature_matrix, axis=0)
    return (feature_matrix / norms, norms)
```

Implementing Coordinate Descent

Theory

We seek to obtain a sparse set of weights by minimizing the LASSO cost function

```
SUM[ (prediction - output)^2 ] + lambda*( |w[1]| + ... + |w[k]|)
```

(By convention, we do not include `w[0]` in the L1 penalty term. We never want to push the intercept to zero.)

The absolute value sign makes the cost function non-differentiable, so simple gradient descent is not viable (you would need to implement a method called subgradient descent). Instead, we will use **coordinate descent**: at each iteration, we will fix all weights but weight `i` and find the value of weight `i` that minimizes the objective. That is, we look for

```
argmin_{w[i]} [ SUM[ (prediction - output)^2 ] + lambda*( |w[1]| + ... + |w
[k]|) ]
```

where all weights other than `w[i]` are held to be constant. We will optimize one `w[i]` at a time, circling through the weights multiple times.

1. Pick a coordinate `i`
2. Compute `w[i]` that minimizes the cost function `SUM[ (prediction - output)^2 ] + lambda*( |w[1]| + ... + |w[k]|)`
3. Repeat Steps 1 and 2 for all coordinates, multiple times

For this notebook, we use **cyclical coordinate descent with normalized features**, where we cycle through coordinates 0 to (d-1) in order, and assume the features were normalized as discussed above. The formula for optimizing each coordinate is as follows:

```
       ┌ (ro[i] + lambda/2)     if ro[i] < -lambda/2
w[i] = ├ 0                      if -lambda/2 <= ro[i] <= lambda/2
       └ (ro[i] - lambda/2)     if ro[i] > lambda/2
```

where

```
ro[i] = SUM[ [feature_i]*(output - prediction + w[i]*[feature_i]) ].
```

Note that we do not regularize the weight of the constant feature (intercept) `w[0]`, so, for this weight, the update is simply:

```
w[0] = ro[i]
```

Effect of L1 penalty

Let us consider a simple model with 2 features:

In [7]:
```
# Get numpy data, normalize, set initial weights, predict output with those weights
(simple_feature_matrix, output) = get_numpy_data(sales, ['sqft_living', 'bedrooms'], 'price')
simple_feature_matrix, norms = normalize_features(simple_feature_matrix)
weights = np.array([1., 4., 1.])
prediction = np.dot(simple_feature_matrix, weights)
```

Compute the values of `ro[i]` for each feature in this simple model, using the formula given above.

In [8]:
```
ro = []
for i in range(len(simple_feature_matrix[0])):
    roItem = (simple_feature_matrix[:,i] * (output - prediction + weights[i] * simple_feature_matrix
[:,i])).sum()
    ro.append(roItem)
    print 'ro for', str(i) + 'th coefficient:', roItem
```

```
ro for 0th coefficient: 79400300.0349
ro for 1th coefficient: 87939470.773
ro for 2th coefficient: 80966698.676
```

QUIZ QUESTION

Whenever `ro[i]` falls between `-l1_penalty/2` and `l1_penalty/2`, the corresponding weight `w[i]` is set to zero. Now suppose we were to take one step of coordinate descent on either feature 1 or feature 2. What range of values of `l1_penalty` **would not** set `w[1]` zero, but **would** set `w[2]` to zero, if we were to take a step in that coordinate? **[161933397.352, 175878941.546]**. What range of values of `l1_penalty` would set **both** `w[1]` and `w[2]` to zero, if we were to take a step in that coordinate? **[175878941.546, ∞[**

In [9]:
```
# Range for first coefficient
l1_penalty_min = -ro[1] * 2
l1_penalty_max = ro[1] * 2
print "Interval of lambda where w[1] is not set to zero: [" + str(l1_penalty_min) + ',', str(l1_penalty_
max)+']'

# Range for second coefficient
l1_penalty_min = -ro[2] * 2
l1_penalty_max = ro[2] * 2
print "Interval of lambda where w[2] is not set to zero: [" + str(l1_penalty_min) + ',', str(l1_penalty_
max)+']'
```

```
Interval of lambda where w[1] is not set to zero: [-175878941.546, 175878941.546]
Interval of lambda where w[2] is not set to zero: [-161933397.352, 161933397.352]
```

So we can say that `ro[i]` quantifies the significance of the i-th feature: the larger `ro[i]` is, the more likely it is for the i-th feature to be retained.

Single Coordinate Descent Step

Using the formula above, implement coordinate descent that minimizes the cost function over a single feature i.

In [10]:
```
def lasso_coordinate_descent_step(i, feature_matrix, output, weights, l1_penalty):
    prediction = np.dot(feature_matrix, weights)
    ro_i = (feature_matrix[:,i] * (output - prediction + weights[i] * feature_matrix[:,i])).sum()

    if i == 0: # intercept -- do not regularize
        new_weight_i = ro_i
    elif ro_i < -l1_penalty/2.:
        new_weight_i = ro_i + l1_penalty / 2
    elif ro_i > l1_penalty/2.:
        new_weight_i = ro_i - l1_penalty / 2
    else:
        new_weight_i = 0.

    return new_weight_i
```

Cyclical coordinate descent

Now that we have a function that optimizes the cost function over a single coordinate, let us implement cyclical coordinate descent where we optimize coordinates 0, 1, ..., (d-1) in order and repeat.

In [11]:
```
def lasso_cyclical_coordinate_descent(feature_matrix, output, weights, l1_penalty, tolerance):
    converged = False
    while not converged:
        change = 0
        for i in range(len(weights)):
            old_weights = weights[i]
            weights[i] = lasso_coordinate_descent_step(i, feature_matrix, output, weights, l1_penalty)
            change += abs(weights[i] - old_weights)
        if change < tolerance:
            converged = True
    return weights
```

Look for model with two inputs.

In [12]:
```
# Get numpy data, normalize
(feature_matrix, output) = get_numpy_data(sales, ['sqft_living', 'bedrooms'], 'price')
(norm_feature_matrix, simple_norms) = normalize_features(feature_matrix)
weights = lasso_cyclical_coordinate_descent(norm_feature_matrix, output, np.zeros(3), 1e7, 1.0)
print "The weight for the constant, sqft_living and bedrooms are ", weights
```

```
The weight for the constant, sqft_living and bedrooms are  [ 21624996.54143479  63157248.454871
24        0.        ]
```

In [13]:
```
prediction = np.dot(norm_feature_matrix, weights)
rss = ((output - prediction) ** 2).sum()
print "The RSS of this model is", rss
```

```
The RSS of this model is 1.63049246479e+15
```

Evaluating LASSO fit with more features

Let us split the sales dataset into training and test sets, and create a normalized feature matrix from the training data with more features.

In [14]:
```
train_data,test_data = sales.random_split(.8,seed=0)

all_features = ['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors', 'waterfront', 'view', 'cond
ition', 'grade', 'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated']
(feature_matrix, output) = get_numpy_data(train_data, all_features, 'price')
(norm_feature_matrix, simple_norms) = normalize_features(feature_matrix)
```

Learn the weights with `l1_penalty = 1e4, 1e7 and 1e8`, on the training data.

In [19]:
```
weights1e4 = lasso_cyclical_coordinate_descent(norm_feature_matrix, output, np.zeros(14), 1e4, 5e
5)
weights1e7 = lasso_cyclical_coordinate_descent(norm_feature_matrix, output, np.zeros(14), 1e7, 1)
weights1e8 = lasso_cyclical_coordinate_descent(norm_feature_matrix, output, np.zeros(14), 1e8, 1)

print "{:>15}".format("Features"), "{:>15}".format("1st Model"), "{:>15}".format("2ed Model"), "{:>
15}".format("3d Model"), "\n"

for i in range(len(weights1e7)):
    if i == 0:
        print "{:>15}".format('constant'), "{:15.0}".format(weights1e4[i]), "{:15.0f}".format(weights1e7[i
]), "{:15.0f}".format(weights1e8[i])
    else:
        print "{:>15}".format(all_features[i-1]), "{:15.0f}".format(weights1e4[i]), "{:15.0f}".format(weigh
ts1e7[i]), "{:15.0f}".format(weights1e8[i])
```

```
    Features      1st Model      2ed Model      3d Model

    constant          1e+08       24429598      71114626
    bedrooms      -19051124              0             0
   bathrooms        3426033              0             0
 sqft_living      173622706       48389176             0
    sqft_lot       -1793050              0             0
      floors       -4477797              0             0
  waterfront        6722093        3317511             0
        view        5760101        7329962             0
   condition       18116054              0             0
       grade       97418322              0             0
  sqft_above      -98542476              0             0
sqft_basement     -26050730              0             0
    yr_built     -188443642              0             0
yr_renovated        3039043              0             0
```

Rescaling learned weights

Create a normalized version of each of the weights learned above. (`weights1e4`, `weights1e7`, `weights1e8`).

In [16]:
```
normalized_weights1e7 = weights1e7 / simple_norms
normalized_weights1e8 = weights1e8 / simple_norms
normalized_weights1e4 = weights1e4 / simple_norms
```

Evaluating each of the learned models on the test data

Let's now evaluate the three models on the test data. Compute the RSS of each of the three normalized weights on the (unnormalized) `test_feature_matrix`:

In [17]:
```
(test_feature_matrix, test_output) = get_numpy_data(test_data, all_features, 'price')
```

In [18]:
```
prediction = np.dot(test_feature_matrix, normalized_weights1e4)
rss1 = ((test_output - prediction) ** 2).sum()
prediction = np.dot(test_feature_matrix, normalized_weights1e7)
rss2 = ((test_output - prediction) ** 2).sum()
prediction = np.dot(test_feature_matrix, normalized_weights1e8)
rss3 = ((test_output - prediction) ** 2).sum()
print "The RSS for our three models are respectively", rss1, rss2, rss3
```

```
The RSS for our three models are respectively 2.06896417446e+14 2.75962075771e+14 5.37166150
034e+14
```