

# Regression For Machine Learning

January 4, 2018

Kyle Hewitt

# 1 Introduction

---

Regression Machine Learning is a way to create predictive models from existing data to predict outcomes of new, never before seen data. This style of machine learning uses existing data to train and access the quality of the predicted model by splitting the existing data into training, validation and testing sets. As we will see when exploring farther into the different regression models, it is very easy to overfit a model to the training data so that the model looks like its perfect for the information its given but when new data is introduced, the predictions can be wildly wrong and sparatic. The idea of Regression models is to use the training data to create a model and then use the validation and testing sets to verify that these models are not overfit and how well they estimate on data outside the training set. There are different error metrics used throughout to test this idea of how well the model fits such as Residual Sum of Squares and RSME. In our experimentation we will be using the gradiant decent and coodinate decent algorithms in order to find the optimal model parameters, but it is worth mentioning that most of the problems addressed can be solved in closed form as well. Techniques such as Linear Regression, Regularization through Ridge and Lasso Regression, and Nearest Neighbor/Kernal Regression will be addressed.

All the code presented here can be referenced at <https://github.com/redonelima/regression>

## 1.1 Notation and software packages used

### 1.1.1 Software Packages Used

Throughout there will usually be two implementations of each idea, one that uses GraphLab and SFrame and another that will be implemented myself. What to expect from this is to see the behavior with these existing tools and then xperimentation in self implementation. Other tools can be used such as syipi and pandas, but this course uses GraphLab because it is good for large datasets because it doesn't require the entire data set to be in memory, instead it only pulls the data into memory as it is needed. This causes the import of GraphLab and the import of the Frame to take a while to load, however, once this is completed operations run faster and large-scale data sets can be used. In the example of the `kc_house_data.gl` set has 17,258 house entries, each with 21 features.

The Numpy library is used for complex matrix calculations. Numpy is a python library which is written in fortran for extremely effcent matrix calculations. These calculations could be done manually in python, but given large data sets, this would be very slow and ineffcent. The MathPlotLib python library is used to visulize data throughout. Graphlab also has functions like `.show()` to visualize data.

The following import will be used throughout most of the code examples thoughout:

```
In [1]: import graphlab
import matplotlib.pyplot as plt
%matplotlib inline
graphlab.canvas.set_target('ipynb', port=None)
sales = graphlab.SFrame('kc_house_data.gl/')
train_data, test_data = sales.random_split(.8, seed=0)
```

This non-commercial license of GraphLab Create for academic use is assigned to khewitt08@live.com and w

[INFO] graphlab.cython.cy\_server: GraphLab Create v2.1 started. Logging: /tmp/graphlab.server.1515119263

## 2 Regression

---

### 2.1 What is Regression?

From a high level perspective, regression is the way to learn about a relationship between *features* and *predictions*. Throughout this case study approach we will be looking at the use-case of predicting housing prices using regressive learning but the scope of regression goes well beyond this, and many of the techniques used throughout the experiments have impacts on other areas of machine learning besides regression such as bias-variance tradeoff which will be covered in depth later on. In the context of regression, the *features* of a model are considered the independant variables and those are derived from the data. For example, for our housing predictive case, the features could be squarefeet, number of bathrooms, lot size, ect. These independant variables are what determine the value of our dependant, *predictive* values, which in our example would be the value or predicted selling price of the home. It is the responsibility of the regressive model to learn the relationship between the independant and dependant variables.

## 3 Linear Regression

---

### 3.1 Simple Linear Regression

The idea of simple linear regression is to fit a line to a dataset. In the simplest form, the regression line is simply a line and intercept. Even if the complexity of the model pushes into a higher order polynomial, the main problem is figuring out which line approximates the best prediction to the test data. In the examples, a dataset of house sales in Kings County Seattle and their features will be used. We start by splitting the dataset into a training set and a test set. We do this because we can fit our model to fit the data perfectly but then any variation in future predictions will be very far from the actual value and the housing price won't be well represented. Therefore we must have a portion of our data serperated from the training set to test the quality of our model.

In a simple linear model, the obvoius equation of the model will be,

$$f(x) = w_0 + w_1x \quad (1)$$

where the equation for the true point can be represented by the equation,

$$y_i = w_0 + w_1x_i + \epsilon \quad (2)$$

where  $\epsilon$  represents the noise that causes variation from the model. This noise will be covered more in depth in the section covering performance assessments.

The next point to address is to figure out how to form a metric to assess the quality of the model. The first way we will address this is called the Residual Sum of Squares (RSS), which is the squared sum of the difference between the real data point and the estimated point given by the prediction model.

$$RSS(w_0, w_1) = \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (3)$$

Where  $\hat{y}_i = w_0 + w_1x_i$  is the value predicted from the model

In the housing example, the most obvious feature to use would be the square footage of the house, represented by the column 'sqft living' in the dataset. In the simple slope-intercept form, the  $w_1$  slope term represents the change in price of the house for every unit of squarefeet.

### 3.2 Gradient Descent

So the next step in finding the lowest error is minimizing the RSS. To do this we have to realize that a plot of every possible fit line will lead to a convex with the optimal point at the minimum. The gradient of a function is the derivative vector in the increasing direction. So in order to find the minumum would be to subtract the gradient. This is known as the gradient decent algorithm. Given some function  $g(w)$ , when we take the derivative with respect to  $w$ , then when  $\frac{dg(w)}{dw}$  is negative, it means we need to increase  $w$  to reach the minimum and when the sign switches it means we need to decrease  $w$ .

This algorithm, known as the gradient decent algorithm, starts at a point and takes steps in the opposite direction of the gradiant. This step size, denoted as  $\eta$  controls how far each iteration of the gradient decent steps towards the minimum. In the case where the derivative of  $g(x)$  is large, then the steps could be very large and may jump back and forth across the minimum and take a very long time to converge. On the other end, if  $\eta$  is too small, then it may take a very long time to converge. This means that step size must be choosen carefully.

There are two popular choices in picking a step size.

1. Fixed step size  
 $\eta = 0.1$

## 2. Decreasing step size

$$\eta_t = \frac{\alpha}{t}$$

$$\eta_t = \frac{\alpha}{\sqrt{t}}$$

In a closed form solution, we would know that we would reach convergence when  $\frac{dg(w)}{dw}$  is zero. Since in a real world situation, with our step size  $\eta$ , it could be impossible to reach zero within a reasonable amount of time. So we have to have looser convergence criteria. Therefore, a threshold criteria  $\varepsilon$  has to be established so that when  $\frac{dg(w)}{dw} < \varepsilon$  we can say we're close enough within convergence to say that we can consider it converged.

This gradient decent algorithm should look like:

$$\text{while } \frac{dg(w)}{dw} \geq \varepsilon:$$

$$w^{(t+1)} = w^t - \eta \frac{dg(w^t)}{dw}$$

The final part of the gradient decent algorithm for our linear regression model is finding  $\frac{dg(w)}{dw}$  where  $g(w_i) = (y_i - w_0 + w_1 x_i)^2$  and since the sum of the derivative is the derivative of the sum we can take the derivative of a single value with respect to  $w_0$  and  $w_1$  to create the gradient vector.

$$\nabla RSS(w_0, w_1) = \begin{bmatrix} -2 \sum_{i=1}^N [y_i - \hat{y}_i] \\ -2 \sum_{i=1}^N [y_i - \hat{y}_i] x_i \end{bmatrix} \quad (4)$$

This ends up being a vector because even in our linear equation we have two variables. This will be a very important concept when we look at multiple regression in the next section where we start adding other features like number of bathrooms or bedrooms ontop of square footage. Given this vector, we can use a closed form solution to solve for each of the unknowns in closed form:

$$\hat{w}_0 = \frac{\sum y_i}{N} - \hat{w}_1 \frac{\sum x_i}{N} \quad (5)$$

$$\hat{w}_1 = \frac{\sum y_i x_i - \frac{\sum y_i}{N} \sum x_i}{\sum x_i^2 - \frac{\sum x_i}{N} \sum x_i} \quad (6)$$

Now that we have these closed form solutions for these two variables. There's a lot of similarities between these two equations so we can pull out a few main calculations that we need, namely:

$$1. \sum y_i \quad 2. \sum x_i \quad 3. \sum y_i x_i \quad 4. \sum x_i^2$$

Using these, we can create a function for calculating simple linear regression.

```
In [2]: def simple_linear_regression(input_feature, output):
    # compute the sum of input_feature and output
    feature_sum = input_feature.sum()
    output_sum = output.sum()
    num_inputs = output.size()

    # compute the product of the output and the input_feature and its sum
    product_sum = (input_feature * output).sum()

    # compute the squared value of the input_feature and its sum
    squared_sum = (input_feature ** 2).sum()
    # use the formula for the slope
    numerator = product_sum - ((float(1)/num_inputs) * (feature_sum*output_sum))
```

```

denominator = squared_sum - ((float(1)/num_inputs) * (feature_sum*feature_sum))
slope = numerator/denominator
# use the formula for the intercept
intercept = output.mean() - slope * input_feature.mean()
return (intercept, slope)

```

Now we can test this function by giving it input that we know the outcome.

We are going to make a feature and then put the output exactly on a line. This results in both our slope and intercept being 1. It will also be used to test our model since all the points will fall on a line it means our RSS will be exactly zero.

```

In [3]: test_feature = graphlab.SArray(range(5))
test_output = graphlab.SArray(1 + 1*test_feature)
(test_intercept, test_slope) = simple_linear_regression(
    test_feature, test_output)
print "Intercept: " + str(test_intercept)
print "Slope: " + str(test_slope)

```

```

Intercept: 1.0
Slope: 1.0

```

As expected, we see that both slope and intercept of our simple test case are 1.

Next, its time to use our simple linear regression model on some actual data. As mentioned earlier our feature for the focus of this model is going to be the square footage of the house and our predictive  $\hat{y}$  is going to be the sales price of the house.

```

In [4]: sqft_intercept, sqft_slope = simple_linear_regression(
    train_data['sqft_living'], train_data['price'])

print "Intercept: " + str(sqft_intercept)
print "Slope: " + str(sqft_slope)

```

```

Intercept: -47116.0765749
Slope: 281.958838568

```

From the above code we can see that we have an intercept approximately  $-47,116$  and a slope of approximately 282.

What this means for us is that a house with no square feet (an empty lot) would actually cost the seller \$47,000 for someone to take. Clearly, this intercept has very little significance for our purposes.

However, the slope value of 281.96 is significant and it means that for every square foot extra a house has, the sales price of the house increases by about \$282. Clearly this is not the most accurate model, since there are many other factors that can play into the value of a home, but that's the reason we're calling this a 'simple' model.

Now that we have a model, we can start making predictions.

### 3.3 Code Example

---

```

In [5]: def get_regression_predictions(input_feature, intercept, slope):
    # calculate the predicted values:
    return slope*input_feature + intercept

```

Now that we have our model and our prediction function we can now use it to predict new houses based on the square footage feature. Lets look at a house that has a size of 2650 sqft.

```
In [6]: my_house_sqft = 2650
        estimated_price = get_regression_predictions(
            my_house_sqft, sqft_intercept, sqft_slope)
        print "The estimated price for a house with %d squarefeet is $%.2f" % (
            my_house_sqft, estimated_price)
```

The estimated price for a house with 2650 squarefeet is \$700074.85

As we can see, using our predictive model a house with 2650 sqft is estimated to be sold for \$ 700,000.

Lets take a look at the quality of our model. We will calculate our RSS, which we already know should be zero.

```
In [7]: def get_residual_sum_of_squares(input_feature, output, intercept, slope):
        # First get the predictions
        predicted_price = get_regression_predictions(input_feature, intercept, slope)
        # then compute the residuals (since we are squaring it doesn't matter which order you subtract)
        residual = output - (slope*input_feature+intercept)
        # square the residuals and add them up
        RSS = residual * residual
        return(RSS.sum())
```

```
In [8]: print get_residual_sum_of_squares(
        test_feature, test_output, test_intercept, test_slope)
        # should be 0.0
```

0.0

```
In [9]: rss_prices_on_sqft = get_residual_sum_of_squares(
        train_data['sqft_living'], train_data['price'],
        sqft_intercept, sqft_slope)
        print 'The RSS of predicting Prices based on Square Feet is : ' + str(
            rss_prices_on_sqft)
```

The RSS of predicting Prices based on Square Feet is : 1.20191835632e+15

We can use this to go the other way as well. Given a price, we can tell how big of a house that a buyer can afford.

```
In [10]: def inverse_regression_predictions(output, intercept, slope):
        return (output-intercept)/slope
```

```
In [11]: my_house_price = 800000
        estimated_squarefeet = inverse_regression_predictions(
            my_house_price, sqft_intercept, sqft_slope)
        print "The estimated squarefeet for a house worth $%.2f is %d" % (
            my_house_price, estimated_squarefeet)
```

The estimated squarefeet for a house worth \$800000.00 is 3004

So we can use this model to say that a buyer with \$800,000 could afford a house up to 3,004sqft.

Now that we have a simple linear model on our housing data, we can change up the features and see how it fits a model when looking at bedrooms and see how that compares to the prediction of square footage.

```
In [12]: bedrooms_intercept, bedrooms_slope = simple_linear_regression(
        train_data['bedrooms'], train_data['price'])

        print "Intercept: " + str(bedrooms_intercept)
        print "Slope: " + str(bedrooms_slope)
```

```
Intercept: 109473.180469
Slope: 127588.952175
```

So based on our bedroom model, we can see that we get an increase in housing price of \$ 127,589 per bedroom. So now that we have our two training models, its time to see how well these predict our test data.

```
In [13]: # Compute RSS when using bedrooms on TEST data:
        rss_prices_on_bedrooms_test = get_residual_sum_of_squares(
            test_data['bedrooms'], test_data['price'],
            bedrooms_intercept, bedrooms_slope)
        print 'The RSS of predicting Prices based on Bedrooms is : ' + str(
            rss_prices_on_bedrooms_test)
```

```
The RSS of predicting Prices based on Bedrooms is : 4.93364582868e+14
```

```
In [14]: # Compute RSS when using squarfeet on TEST data:
        rss_prices_on_sqft_test = get_residual_sum_of_squares(
            test_data['sqft_living'], test_data['price'],
            sqft_intercept, sqft_slope)
        print 'The RSS of predicting Prices based on Square Feet is : ' + str(
            rss_prices_on_sqft_test)
```

```
The RSS of predicting Prices based on Square Feet is : 2.75402936247e+14
```

Now that we have our RSS of our squarefoot model and our bedroom model, we can compare our two RSS and see that since the squarefootage model has a lower RSS on the testing data it does a better job of a predictive model. This is because we trained our model on the training data, so all of the houses in the testing set are all houses that the model has never seen. Since the RSS of squarefeet is lower, it means that the houses in the testing with the squarefeet feature set were all closer to the prediction model than those of the bedroom feature set.

It is important at this point to address the fact that this model is assuming an asymmetric error, meaning that we're assuming the same consequence for under estimating than over estimating. With housing, this may not be the case. A house whose value is over estimated may not get many offers and therefore take longer to sell. A house that is underestimated may be listed too low and the buyer ends up losing money. The consequences of each of these errors are actually asymmetric, but this adds an entirely new level of complexity so we will be treating errors symmetrically.



## 4 Multiple Regression

Anyone who has ever had to deal with valuing a home knows there's a lot more to it than just the square footage or the number of bedrooms by themselves. In this section we will be looking at multiple regression, meaning that we're going to be building a regression model based off of multiple features and not just off a single one. We will also be looking at what is known as polynomial regression which will allow us to fit our model to a higher order polynomial, giving our model much more flexibility.

There are not always exponential relationships between features and their trends. An example of a polynomial equation could be  $y_i = w_0 + w_1t_i + w_2t_i^2 + \dots + w_nt_i^n$  where as a more complex model that has a linearly upward trend could look like  $y_i = w_0 + w_1t_i + w_2 \sin(\frac{2\pi t_i}{12} - \phi) + \epsilon_i$

This second equation represents seasonality in a linearly upward trend, which is a good representative model of anything that has varying trends throughout the year but has an overall upward trend. Examples of this are camping equipment sales that have higher sales throughout the summer during the camping season. Clearly this will give an overall better prediction than a linear model, but this is much harder to model. This is where the power of regressive learning and gradient decent start to really show their power.

### 4.1 The Polynomial Regression Model

We are going to look at this in general form, also known as the generic basis expansion. Each  $j^{th}$  feature will have a bias, as seen with the example of the sin function above, which we will refer to as  $h_j(x_i)$  which is the  $j^{th}$  feature of data input  $i$  referred to as  $x_i$ .

So the generic basis expansion for each  $x_i$  data input:

$$\sum_{j=0}^D w_j h_j(x_i) + \epsilon_i \quad (7)$$

Where  $D$  is the number of features for each data point

This means that instead of being in a 2 dimensional space like we were when we were looking at a single feature, we will be looking at a  $D+1$  dimensional space when looking at multiple features. So for a model that accounts for 2 features, we'd be looking at a hyperplane in a 3D space instead of a line in a 2D space. This is also known as a  $D$ -dimensional curve.

So how do we look at this regression model in this new  $D$ -dimensional space? We have to start looking at the data points in matrix notation since we're dealing with an  $n \times m$  matrix of observations which we will refer to as  $\mathbf{H}$  where  $n$  is the number of features  $D$  and  $m$  is the number of observations  $i$ .

$\mathbf{H}$  is multiplied by the  $D \times 1$  feature vector  $\vec{w}$ . Finally we add in our error vector  $\vec{\epsilon}$ . Finally this will give us our final vector  $\vec{y}$ .

$$\vec{y} = \mathbf{H}\vec{w} + \vec{\epsilon} \quad (8)$$

We will start by looking at an instance of  $D$  observations, each with a single feature, like in our linear regression model. because of the rules of vector multiplication, we will take the transpose of this observation/feature vector, denoted  $\vec{h}$ . This is then multiplied by the coefficient vector  $\vec{w}$  and added to our single  $\epsilon$  value.

$$y_i = \vec{h}^T(x_i)\vec{w} + \epsilon \quad (9)$$

So our vectors for a single iteration would look like:

$$y_i = [w_0 \quad w_1 \quad w_2 \quad \dots \quad w_D] + \begin{bmatrix} h_0(x_i) \\ h_1(x_i) \\ h_2(x_i) \\ \vdots \\ h_D(x_i) \end{bmatrix} + [\epsilon] \quad (10)$$

So lets look at this as a matrix of  $N$  observations:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} h_0(x_1) & h_1(x_1) & \dots & h_D(x_1) \\ h_0(x_2) & h_1(x_2) & \dots & h_D(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ h_0(x_N) & h_1(x_N) & \dots & h_D(x_N) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_D \end{bmatrix} + \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \vdots \\ \epsilon_N \end{bmatrix} \quad (11)$$

## 4.2 Computing the RSS

Now that we have our multiple feature regression model, we need to figure out how we're going to assess the cost of a given model in order to find the one that best fits our predictive model. Just like we did in our simple linear model, we will use RSS as a metric for this cost. Just like with our model we will write the RSS in terms of a single component and then scale it up. Since we know that the RSS is the difference between the actual value and the predicted value, our first part of the RSS equation will remain the same. What will change is the definition of our  $\hat{y}$  value. We will write this in terms of the model itself this time instead of in terms of  $\hat{y}$  because it will be easier to scale it up into matrix form.

$$RSS(\vec{w}) = \sum_{i=1}^N (y_i - \vec{h}^T(x_i)\vec{w})^2 \quad (12)$$

Since we know that  $\vec{h}(x_i)$  is a single component of  $\mathbf{H}$ , then the matrix form of  $\vec{h}(x_i)\vec{w}$  would be  $\mathbf{H}\vec{w}$  and to square each component in the resulting vector we would multiply the resulting vector by its transpose. Therefore our matrix notation of the RSS would be:

$$RSS(\vec{w}) = \sum_{i=1}^N (y - \mathbf{H}\vec{w})^T (y - \mathbf{H}\vec{w}) \quad (13)$$

## 4.3 Gradient of the RSS

Now that we have our model for our RSS, we need to find the gradient in order to find our lowest cost model. The gradient is much more useful in this polynomial regression model than it did in the linear model because now we have more features. To find our gradient we will take the same steps as we did in earlier derivations and solve for our gradient in 1D and scale it up. To find the derivative of our 1D case we must note that both  $y$ ,  $h$  and  $w$  are scalars in our 1D case.

$$\begin{aligned} \frac{d}{dw}(y - hw)(y - hw) &= \frac{d}{dw}(y - hw)^2 \\ &= 2(y - hw)^1(-h) \\ &= -2h(y - hw) \end{aligned} \quad (14)$$

So its not hard to see that when its scaled up our gradient is:

$$\begin{aligned} \nabla RSS(\vec{w}) &= \nabla[(\vec{y} - \mathbf{H}\vec{w})^T(\vec{y} - \mathbf{H}\vec{w})] \\ &= -2\mathbf{H}^T(\vec{y} - \mathbf{H}\vec{w}) \end{aligned} \quad (15)$$

### 4.3.1 Closed Form Solution

Just like in our linear model, we can solve for our polynomial model in the same closed form way. Now that we have our gradient of our RSS, we can set it equal to zero and solve for  $\hat{w}$

$$\begin{aligned}
-2\mathbf{H}^T(\vec{y} - \mathbf{H}\vec{w}) &= 0 \\
-2\mathbf{H}^T\vec{y} + 2\mathbf{H}^T\mathbf{H}\vec{w} &= 0 \\
\mathbf{H}^T\mathbf{H}\vec{w} &= \mathbf{H}^T\vec{y} \\
(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{H}\vec{w} &= (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\vec{y} \\
\mathbf{I}\vec{w} &= (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\vec{y} \\
\vec{w} &= (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\vec{y}
\end{aligned} \tag{16}$$

This solved out very smoothly but there's a few problems when approaching this in a closed form way like we did with the linear model. Both of the main issues have to do with the inverse. First,  $(\mathbf{H}^T\mathbf{H})^{-1}$  is only invertable if, in most cases, the number of linearly independant observations  $N > D$ . This limits the complexity of the models we can make. This issue is addressed later with regularization. The second issue is the complexity of doing the inverse. The complexity of inverting a  $D \times D$  matrix is  $\Omega(D^3)$  which can get very computationally expensive with large data sets. Therefore using this method is only an option on reasonably small sets of data.

### 4.3.2 Gradient Descent

The gradient descent for polynomial regression isn't much different from linear regression, just accounting for more features. The first difference is the convergence criteria. Now that we have many other features accounted for in the gradient, now we must find the minimum of each of those features. The value cost of each feature is in the RSS vector. Therefore the convergence of the function would be when the magnitude of the gradient of the RSS is zero, or within the threshold  $\varepsilon$  in our real world case. More formally,  $\|\nabla RSS(\vec{w}^{(t)})\| > \varepsilon$

Then for each of the features, referred to as partials here, the sum of the dirivitive of the partials are multiplied by the step size  $\eta$  for each of the iterations  $t$ .

```

while  $\|\nabla RSS(\vec{w}^t)\| > \varepsilon$  :
    for j in range(0, D):
         $partial_j = -2 \sum_{i=1}^N h_j(\vec{x}_i)(y_i - \hat{y}_i(\vec{w}^{(t)}))$ 
         $\vec{w}_j^{(t+1)} = \vec{w}_j^t - \eta(partial_j)$ 
t++

```

## 4.4 Code Example

---

We will start out with using the GraphLab linear\_regression model again. This time however we will be passing it a list of features which will cause it to run polynomial regression.

```

In [15]: example_features = ['sqft_living', 'bedrooms', 'bathrooms']
         example_model = graphlab.linear_regression.create(
             train_data, target = 'price', features = example_features,
                                     validation_set = None)

```

Linear regression:

-----

Number of examples : 17384

Number of features : 3

Number of unpacked features : 3

Number of coefficients : 4

Starting Newton Method

```
-----
```

Iteration	Passes	Elapsed Time	Training-max_error	Training-rmse
1	2	1.068897	4146407.600631	258679.804477

```
-----
```

SUCCESS: Optimal solution found.

We now have a trained model. From here we can look at the coefficients of the model and make predictions based off the data.

```
In [16]: example_weight_summary = example_model.get("coefficients")
         print example_weight_summary
         example_predictions = example_model.predict(train_data)
         print example_predictions[0] # should be 271789.505878
```

```
-----
```

name	index	value	stderr
(intercept)	None	87910.0724924	7873.3381434
sqft_living	None	315.403440552	3.45570032585
bedrooms	None	-65080.2155528	2717.45685442
bathrooms	None	6944.02019265	3923.11493144

```
-----
```

[4 rows x 4 columns]

271789.505878

Now that we have a trained model we can compute the cost metric using RSS.

```
In [17]: def get_residual_sum_of_squares(model, data, outcome):
        # First get the predictions
        predictions = model.predict(data)
        # Then compute the residuals/errors
        errors = outcome - predictions
        # Then square and add them up
        RSS = (errors ** 2).sum()
        return(RSS)

In [18]: rss_example_train = get_residual_sum_of_squares(example_model, test_data, test_data['price'])
        print rss_example_train # should be 2.7376153833e+14
```

2.7376153833e+14

Now that we've verified that our model is trained, we can add a few more features based off of the features we already have. The reason for this is that features are sometimes codependant and more accurate models can be created by a combination of other features. For this experiment we will be the square of the bedroom count, the product of bedrooms and bathrooms, the log of the square feet, and the sum of the properties latitude and longitude.

```
In [19]: from math import log
        train_data['bedrooms_squared'] = train_data['bedrooms'].apply(lambda x: x**2)
        train_data['bed_bath_rooms'] = train_data['bedrooms'] * train_data['bathrooms']
        train_data['log_sqft_living'] = train_data['sqft_living'].apply(lambda x: log(x))
        train_data['lat_plus_long'] = train_data['lat'] + train_data['long']

        test_data['bedrooms_squared'] = test_data['bedrooms'].apply(lambda x: x**2)
        test_data['bed_bath_rooms'] = test_data['bedrooms'] * test_data['bathrooms']
        test_data['log_sqft_living'] = test_data['sqft_living'].apply(lambda x: log(x))
        test_data['lat_plus_long'] = test_data['lat'] + test_data['long']
```

The purpose of squaring the bedrooms is to separate data more between few bedrooms and many bedrooms. The product of the bedrooms and bathrooms is known as an interaction feature. It is large only when both bedrooms and bathroom counts are large. Taking the log of the square feet will spread out small values and bring larger values closer together. The final feature of the sum of the latitude and longitude is only to show that using poor feature selection leads to poor predictive models.

For this experiment, we will make three separate models. The first will have bedrooms, bathrooms, square feet, latitude, and latitude. All of these features exist in the original dataset.

The next model will include all of the features of model 1 plus the remainder of the new features we just created.

```
In [20]: model_1_features = ['sqft_living', 'bedrooms', 'bathrooms', 'lat', 'long']
        model_2_features = model_1_features + ['bed_bath_rooms']
        model_3_features = model_2_features + ['bedrooms_squared', 'log_sqft_living', 'lat_plus_long']

In [21]: model_1 = graphlab.linear_regression.create(train_data, target = 'price',
                                                    features = model_1_features, validation_set = None)
        model_2 = graphlab.linear_regression.create(train_data, target = 'price',
                                                    features = model_2_features, validation_set = None)
        model_3 = graphlab.linear_regression.create(train_data, target = 'price',
                                                    features = model_3_features, validation_set = None)
```

Linear regression:

-----

Number of examples : 17384

Number of features : 5

Number of unpacked features : 5

Number of coefficients : 6

Starting Newton Method

-----				
+-----+-----+-----+-----+-----+				
Iteration	Passes	Elapsed Time	Training-max_error	Training-rmse
+-----+-----+-----+-----+-----+				
1	2	0.098199	4074878.213096	236378.596455
+-----+-----+-----+-----+-----+				

SUCCESS: Optimal solution found.

Linear regression:

-----

Number of examples : 17384

Number of features : 6

Number of unpacked features : 6

Number of coefficients : 7

Starting Newton Method

-----				
+-----+-----+-----+-----+-----+				
Iteration	Passes	Elapsed Time	Training-max_error	Training-rmse
+-----+-----+-----+-----+-----+				
1	2	0.132620	4014170.932927	235190.935428
+-----+-----+-----+-----+-----+				

SUCCESS: Optimal solution found.

Linear regression:

-----

Number of examples : 17384

Number of features : 9

Number of unpacked features : 9

Number of coefficients : 10

Starting Newton Method

-----				
+-----+-----+-----+-----+-----+				
Iteration	Passes	Elapsed Time	Training-max_error	Training-rmse
+-----+-----+-----+-----+-----+				
1	2	0.107355	3193229.177894	228200.043155

-----+

SUCCESS: Optimal solution found.

```
In [22]: # Examine/extract each model's coefficients:
print model_1.get("coefficients")
print model_2.get("coefficients")
print model_3.get("coefficients")
```

```
-----+
|      name      | index |      value      |      stderr      |
-----+
| (intercept)    | None  | -56140675.7444  | 1649985.42028    |
| sqft_living     | None  | 310.263325778   | 3.18882960408    |
| bedrooms       | None  | -59577.1160682  | 2487.27977322    |
| bathrooms      | None  | 13811.8405418   | 3593.54213297    |
| lat            | None  | 629865.789485   | 13120.7100323    |
| long           | None  | -214790.285186  | 13284.2851607    |
-----+
[6 rows x 4 columns]
```

```
-----+
|      name      | index |      value      |      stderr      |
-----+
| (intercept)    | None  | -54410676.1152  | 1650405.16541    |
| sqft_living     | None  | 304.449298057   | 3.20217535637    |
| bedrooms       | None  | -116366.043231  | 4805.54966546    |
| bathrooms      | None  | -77972.3305135  | 7565.05991091    |
| lat            | None  | 625433.834953   | 13058.3530972    |
| long           | None  | -203958.60296   | 13268.1283711    |
| bed_bath_rooms | None  | 26961.6249092   | 1956.36561555    |
-----+
[7 rows x 4 columns]
```

```
-----+
|      name      | index |      value      |      stderr      |
-----+
| (intercept)    | None  | -52974974.0602  | 1615194.9439     |
| sqft_living     | None  | 529.196420564   | 7.69913498511    |
| bedrooms       | None  | 28948.5277313   | 9395.72889106    |
| bathrooms      | None  | 65661.207231    | 10795.3380703    |
| lat            | None  | 704762.148408   | 1292011141.66    |
| long           | None  | -137780.01994   | 1292011141.57    |
| bed_bath_rooms | None  | -8478.36410518  | 2858.95391257    |
| bedrooms_squared | None  | -6072.38466067  | 1494.97042777    |
| log_sqft_living | None  | -563467.784269  | 17567.8230814    |
| lat_plus_long   | None  | -83217.1979248  | 1292011141.58    |
-----+
[10 rows x 4 columns]
```



Now that we have the coefficients of our three models, we can compare what impacts the inclusion of new features has. We see a change in sign from the first model to the second, which tells us that when we add the interaction feature to the set, the bedroom coefficient goes the opposite direction.

Now that we have our two models, we can compute the RSS to find how well our models represent the real. We will first compute the RSS on our training data, which we can expect to be very similar, and the more complex model will have the lower training error, intuitively. Then we will run the RSS on the test set and see which model shows the best on data that it was not trained on.

```
In [23]: # Compute the RSS on TRAINING data for each of the three models and record the values:
RSS_1 = get_residual_sum_of_squares(model_1, train_data, train_data['price'])
RSS_2 = get_residual_sum_of_squares(model_2, train_data, train_data['price'])
RSS_3 = get_residual_sum_of_squares(model_3, train_data, train_data['price'])

print RSS_1
print RSS_2
print RSS_3

9.71328233544e+14
9.61592067856e+14
9.05276314555e+14
```

Like expected, our RSS was lowest on the third model because it had the most features.

```
In [24]: # Compute the RSS on TESTING data for each of the three models and record the values:
RSS_1 = get_residual_sum_of_squares(model_1, test_data, test_data['price'])
RSS_2 = get_residual_sum_of_squares(model_2, test_data, test_data['price'])
RSS_3 = get_residual_sum_of_squares(model_3, test_data, test_data['price'])

print RSS_1
print RSS_2
print RSS_3

2.26568089093e+14
2.24368799994e+14
2.51829318952e+14
```

Now we can see that the second model actually represented the best model with the lowest RSS on the test set. When we gave it more meaningful features, the predictive model improved and the test data fell closest to the predicted model. However, in the third model we added three more, less meaningful features (including one meaningless feature) and we see that it actually has the worst performing model. Feature selection is clearly a very important aspect of regression learning and will be covered in depth in a later section in regularization.

Now that we see how this polynomial regression behaves in GraphLab, let's implement some polynomial regression ourselves. Since we will be using numpy for our matrix calculations, we need to have a function that converts an SFrame into a numpy array.

```
In [31]: import numpy as np

(example_features, example_output) = get_numpy_data(sales, ['sqft_living'], 'price')

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 extract it all
    features = ['constant'] + features # this is how you combine two lists
```

```

        # select the columns of data_SFrame given by the features list into the SFrame features_sf
        features_sframe = data_sframe[features]
        # 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 output_sarray
        output_sarray = data_sframe[output]
        # the following will convert the SArray into a numpy array by first converting it to a list
        output_array = output_sarray.to_numpy()
        return(feature_matrix, output_array)

def predict_output(feature_matrix, weights):
    return np.dot(feature_matrix, weights)

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 ' + str(predicted_value)

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

predicted value 1181.0
1181.0
2571.0

```

```

In [32]: def feature_derivative(errors, feature):
        # Assume that errors and feature are both numpy arrays of the same length (number of data points)
        # compute twice the dot product of these vectors as 'derivative' and return the value
        derivative = 2 * np.dot(errors, feature)
        return(derivative)

In [33]: (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 the -example_output
feature = example_features[:,0] # let's compute the derivative with respect to 'constant', the intercept
derivative = feature_derivative(errors, feature)
print derivative
print -np.sum(example_output)*2 # should be the same as derivative

-23345850022.0
-23345850022.0

```

Now that we've created all of our functions we need and computed our derivatives, its time to implement our gradient descent algorithm.

```

In [38]: from math import sqrt
def regression_gradient_descent(feature_matrix, output, initial_weights, step_size, tolerance):
    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 predict_output function
        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 weights[i]
    # 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 assessing
    gradient_sum_squares = gradient_sum_squares + (derivative * derivative)
    # subtract the step size times the derivative from the current weight
    weights[i] = weights[i] - (step_size * derivative)
# compute the square-root of the gradient sum of squares to get the gradient magnitude
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 and the output is large. So while you might expect tolerance to be small, small is only relative to the size of the features.

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

```

In [39]: # 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_output)
initial_weights = np.array([-47000., 1.])
step_size = 7e-12
tolerance = 2.5e7

simple_weights = regression_gradient_descent(simple_feature_matrix, output, initial_weights, s
print simple_weights
print round(simple_weights[1], 1)

```

```

[-46999.88716555    281.91211912]
281.9

```

```

In [ ]:

```