SUMMER TRAINING REPORT

On

MACHINE LEARNING - REGRESSION

Submitted to Guru Gobind Singh Indraprastha University, Delhi (India)
in partial fulfillment of the requirement for the award of the degree of

B.TECH

in

ELECTRONICS AND COMMUNICATIONS ENGINEERING

Submitted By
MAYANK CHOUDHARY

Roll. No. 35515002818

DEPTT. OF ELECTRONICS AND COMMUNICATIONS

MAHARAJA SURAJMAL INSTITUTE OF TECHNOLOGY ,
NEW DELHI-110058

AUGUST 2020

ACKNOWLEDGEMENT

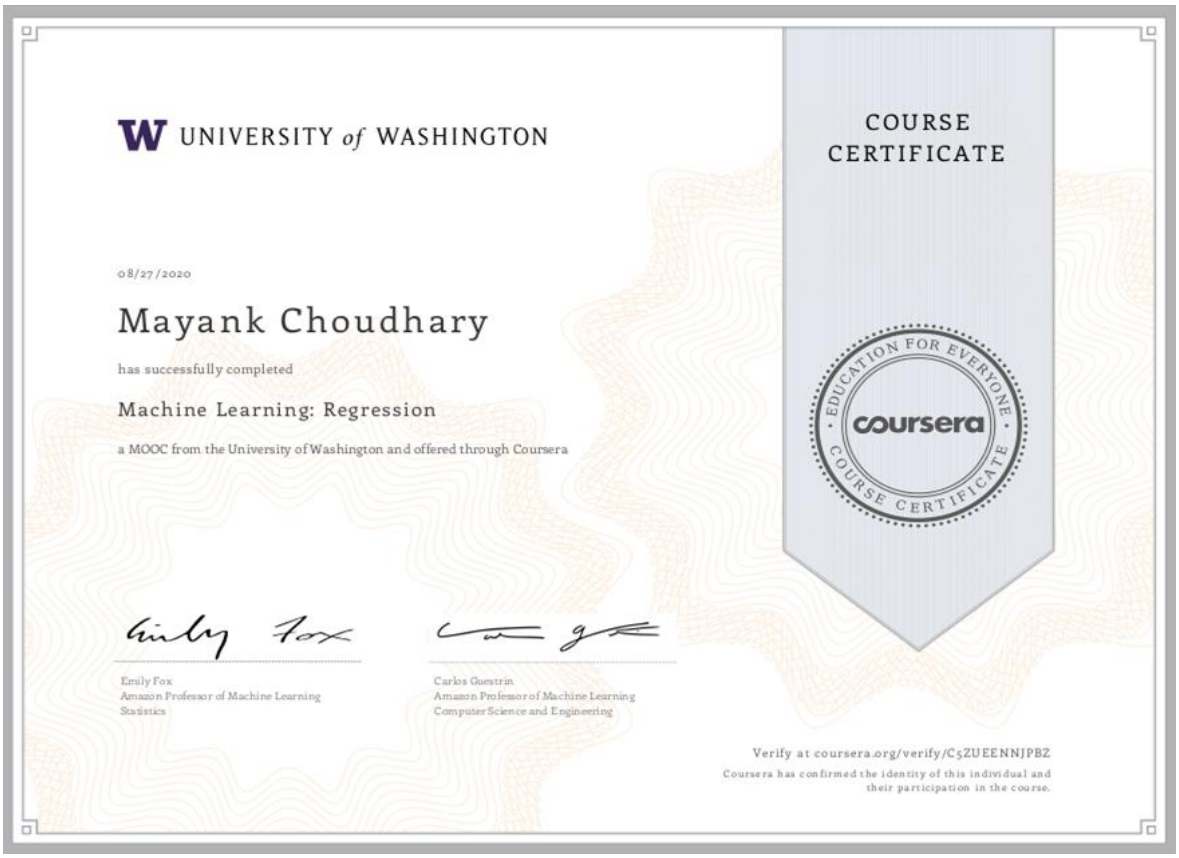
A research work owes its success from commencement to completion, to the people in love with researchers at various stages. Let me in this page express my gratitude to all those who helped us in various stage of this study. First, I would like to express my sincere gratitude indebtedness to **Mr. PUNEET AZAD** (HOD, Department of Information Technology, Maharaja Surajmal Institute of Technology,New delhi) for allowing me to undergo the summer training of 30 days at …….....**COURSERA**.....

I am grateful to my instructors **Carlos Guestrin & Emily Fox** for this easy to learn course and their way of teaching marks a really strong effect.

Last but not least, I pay my sincere thanks and gratitude to all the Staff Members of Maharaja Surajmal Institute of Technology,New delhi for their support and for making our training valuable and fruitful.

Submitted By: Mayank Choudhary

CERTIFICATE



CANDIDATE'S DECLARATION

I **MAYANK CHOUDHARY** , Roll No **35515002818**, B.Tech (Semester- 5^{th}) of the Maharaja Surajmal Institute of Technology, New Delhi hereby declare that the Training Report entitled **" MACHINE LEARNING - REGRESSION"** is an original work and data provided in the study is authentic to the best of my knowledge. This report has not been submitted to any other Institute for the award of any other degree.

Name of Student : Mayank Choudhary
(Roll No: 35515002818**)**

Organization Introduction

Coursera is a world-wide online learning platform founded in 2012 by Stanford University's computer science professors Andrew Ng and Daphne Koller that offers massive open online courses, specializations, degrees, professional and mastertrack courses.

The course is offered by University of Washington

Instructors -

Emily Fox

Amazon Professor of Machine Learning
- Statistics

Carlos Guestrin

Amazon Professor of Machine Learning
- Computer Science and Engineering

List of Tables

List of Figures

Contents

References

- Coursera online training course
- Google Images

Simple Linear Regression

Our course starts from the most basic regression model: Just fitting a line to data. This simple model for forming predictions from a single, univariate feature of the data is appropriately called "simple linear regression". In this module, we describe the high-level regression task and then specialize these concepts to the simple linear regression case. We learn how to formulate a simple regression model and fit the model to data using both a closed-form solution as well as an iterative optimization algorithm called gradient descent. Based on this fitted function, we will interpret the estimated model parameters and form predictions. We will also analyze the sensitivity of our fit to outlying observations. We will then examine all of these concepts in the context of a case study of predicting house prices from the square feet of the house.

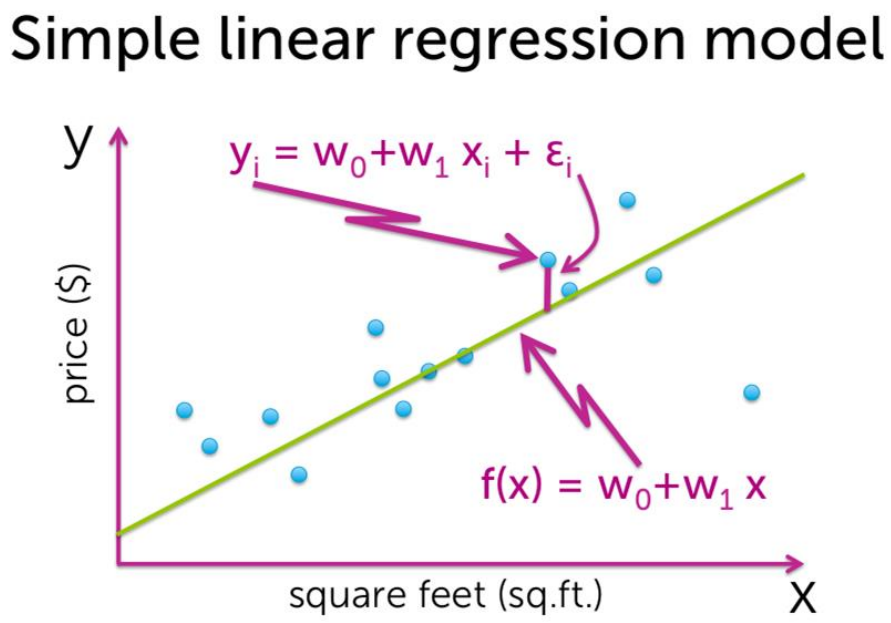


Fig.1

Regression Week 1:

In this notebook we will use data on house sales in King County to predict house prices using simple (one input) linear regression. You will:

Use Turi Create SArray and SFrame functions to compute important summary statistics

Write a function to compute the Simple Linear Regression weights using the closed form solution

Write a function to make predictions of the output given the input feature

Turn the regression around to predict the input given the output

Compare two different models for predicting house prices

In this notebook you will be provided with some already complete code as well as some code that you should complete yourself in order to answer quiz questions. The code we provide to complete is optional and is there to assist you with solving the problems but feel free to ignore the helper code and write your own.

Fire up Turi Create

```
import turicreate
```

Load house sales data

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

```
sales = turicreate.Sframe('home_data.sframe')
```

Split data into training and testing

We use seed=0 so that everyone running this notebook gets the same results. In practice, you may set a random seed (or let Turi Create pick a random seed for you).

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

Useful SFrame summary functions

In order to make use of the closed form solution as well as take advantage of turi create's built in functions we will review some important ones. In particular:

Computing the sum of an SArray

Computing the arithmetic average (mean) of an SArray

multiplying SArrays by constants

multiplying SArrays by other SArrays

```
# Let's compute the mean of the House Prices in King County in 2 different ways.
```

```
prices = sales['price'] # extract the price column of the sales SFrame -- this is now an SArray

# recall that the arithmetic average (the mean) is the sum of the prices divided by the total number of houses:
sum_prices = prices.sum()
num_houses = len(prices) # when prices is an SArray len() returns its length
avg_price_1 = sum_prices/num_houses
avg_price_2 = prices.mean() # if you just want the average, the .mean() function
print (("average price via method 1: ") + str(avg_price_1))
print (("average price via method 2: ") + str(avg_price_2))
average price via method 1: 540088.1419053348
average price via method 2: 540088.1419053345
```

As we see we get the same answer both ways

```
# if we want to multiply every price by 0.5 it's a simple as:
half_prices = 0.5*prices
# Let's compute the sum of squares of price. We can multiply two SArrays of the same length elementwise also with *
prices_squared = prices*prices
sum_prices_squared = prices_squared.sum() # price_squared is an SArray of the squares and we want to add them up.
print (("the sum of price squared is: ") + str(sum_prices_squared))
the sum of price squared is: 9217325133550736.0
```

Aside: The python notation x.xxe+yy means x.xx * 10^(yy). e.g 100 = 10^2 = 1*10^2 = 1e2

1.1 Build a generic simple linear regression function

Armed with these SArray functions we can use the closed form solution found from lecture to compute the slope and intercept for a simple linear regression on observations stored as SArrays: input_feature, output.

Complete the following function (or write your own) to compute the simple linear regression slope and intercept:

```
def simple_linear_regression(input_feature, output):
    # compute the sum of input_feature and output

    # compute the product of the output and the input_feature and its sum

    # compute the squared value of the input_feature and its sum

    # use the formula for the slope

    # use the formula for the intercept
    num= (input_feature*output).mean()- (input_feature.mean())*(output.mean())
    den= ((input_feature*input_feature).mean())-(input_feature.mean())*(input_feature.mean())

    slope= num/den

    intercept=(output.mean())- slope*(input_feature.mean())


    return (intercept, slope)
```

We can test that our function works by passing it something where we know the answer. In particular we can generate a feature and then put the output exactly on a line: output = 1 + 1*input_feature then we know both our slope and intercept should be 1

```
test_feature = turicreate.SArray(range(5))
test_output = turicreate.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

Now that we know it works let's build a regression model for predicting price based on sqft_living. Rembember that we train on train_data!

```
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: -47116.076574944775
Slope: 281.9588385676992 ,,,,,,,,,,,,,,,,,, (1)

Predicting Values

Now that we have the model parameters: intercept & slope we can make predictions. Using SArrays it's easy to multiply an SArray by a constant and add a constant value. Complete the following function to return the predicted output given the input_feature, slope and intercept:

```
def get_regression_predictions(input_feature, intercept, slope):
    # calculate the predicted values:
    predicted_values= (input_feature*slope)+intercept
    return predicted_values
```

Now that we can calculate a prediction given the slope and intercept let's make a prediction. Use (or alter) the following to find out the estimated price for a house with 2650 squarefeet according to the squarefeet model we estiamted above.

Quiz Question: Using your Slope and Intercept from (1), What is the predicted price for a house with 2650 sqft?

```
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

1.2 Residual Sum of Squares

Now that we have a model and can make predictions let's evaluate our model using Residual Sum of Squares (RSS). Recall that RSS is the sum of the squares of the residuals and the residuals is just a fancy word for the difference between the predicted output and the true output.

Complete the following (or write your own) function to compute the RSS of a simple linear regression model given the input_feature, output, intercept and slope:

```
def get_residual_sum_of_squares(input_feature, output, intercept, slope):
    y_hat = intercept + input_feature*slope
    return ((output-y_hat)**2).sum()
    # First get the predictions

    # then compute the residuals (since we are squaring it doesn't matter which order you subtract)

    # square the residuals and add them up
```

Let's test our get_residual_sum_of_squares function by applying it to the test model where the data lie exactly on a line. Since they lie exactly on a line the residual sum of squares should be zero!

```
print (get_residual_sum_of_squares(test_feature, test_output, test_intercept, test_slope)) # should be 0.0
0.0
```

Now use your function to calculate the RSS on training data from the squarefeet model calculated above.

Quiz Question: According to this function and the slope and intercept from the squarefeet model What is the RSS for the simple linear regression using squarefeet to predict prices on TRAINING data?

```
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 : 1201918356321968.2
```

Predict the squarefeet given price

What if we want to predict the squarefoot given the price? Since we have an equation y = a + b*x we can solve the function for x. So that if we have the intercept (a) and the slope (b) and the price (y) we can solve for the estimated squarefeet (x).

Complete the following function to compute the inverse regression estimate, i.e. predict the input_feature given the output.

```
def inverse_regression_predictions(output, intercept, slope):
    # solve output = intercept + slope*input_feature for input_feature. Use this equation to compute the inverse predictions:
    estimated_feature = (output - intercept)/float(slope)
    return estimated_feature
```

Now that we have a function to compute the squarefeet given the price from our simple regression model let's see how big we might expect a house that costs $800,000 to be.

Quiz Question: According to this function and the regression slope and intercept from (3) what is the estimated square-feet for a house costing $800,000?

```
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
```

New Model: estimate prices from bedrooms

We have made one model for predicting house prices using squarefeet, but there are many other features in the sales SFrame. Use your simple linear regression function to estimate the regression parameters from predicting Prices based on number of bedrooms. Use the training data!

```
# Estimate the slope and intercept for predicting 'price' based on 'bedrooms'

bedroom_intercept, bedroom_slope = simple_linear_regression(train_data['bedrooms'], train_data['price'])
print ("Bedroom intercept: ", bedroom_intercept)
print ("Bedroom slope: ", bedroom_slope)
```

```
Bedroom intercept:  109473.18046925141
Bedroom slope:  127588.95217459423
```

Test your Linear Regression Algorithm

Now we have two models for predicting the price of a house. How do we know which one is better? Calculate the RSS on the TEST data (remember this data wasn't involved in learning the model). Compute the RSS from predicting prices using bedrooms and from predicting prices using squarefeet.

Quiz Question: Which model (square feet or bedrooms) has lowest RSS on TEST data? Think about why this might be the case.

```
# Compute RSS when using bedrooms on TEST data:
rss_prices_on_bedrooms = get_residual_sum_of_squares(test_data['bedrooms'], test_data['price'],
sqft_intercept, sqft_slope)
print ('The RSS of predicting Prices based on bedrooms is : ' + str(rss_prices_on_bedrooms))
```

```
The RSS of predicting Prices based on bedrooms is : 2005068534224686.2
```

```
sales
```

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
7129300520	2014-10-13 00:00:00+00:00	221900.0	3.0	1.0	1180.0	5650.0	1.0	0
6414100192	2014-12-09 00:00:00+00:00	538000.0	3.0	2.25	2570.0	7242.0	2.0	0

5631500400	2015-02-25 00:00:00+00:00	180000.0	2.0	1.0	770.0	10000.0	1.0	0
2487200875	2014-12-09 00:00:00+00:00	604000.0	4.0	3.0	1960.0	5000.0	1.0	0
1954400510	2015-02-18 00:00:00+00:00	510000.0	3.0	2.0	1680.0	8080.0	1.0	0
7237550310	2014-05-12 00:00:00+00:00	1225000.0	4.0	4.5	5420.0	101930.0	1.0	0
1321400060	2014-06-27 00:00:00+00:00	257500.0	3.0	2.25	1715.0	6819.0	2.0	0
2008000270	2015-01-15 00:00:00+00:00	291850.0	3.0	1.5	1060.0	9711.0	1.0	0
2414600126	2015-04-15 00:00:00+00:00	229500.0	3.0	1.0	1780.0	7470.0	1.0	0
3793500160	2015-03-12 00:00:00+00:00	323000.0	3.0	2.5	1890.0	6560.0	2.0	0

view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat
0	3	7.0	1180.0	0.0	1955.0	0.0	98178	47.51123398
0	3	7.0	2170.0	400.0	1951.0	1991.0	98125	47.72102274
0	3	6.0	770.0	0.0	1933.0	0.0	98028	47.73792661
0	5	7.0	1050.0	910.0	1965.0	0.0	98136	47.52082
0	3	8.0	1680.0	0.0	1987.0	0.0	98074	47.61681228
0	3	11.0	3890.0	1530.0	2001.0	0.0	98053	47.65611835
0	3	7.0	1715.0	0.0	1995.0	0.0	98003	47.30972002
0	3	7.0	1060.0	0.0	1963.0	0.0	98198	47.40949984
0	3	7.0	1050.0	730.0	1960.0	0.0	98146	47.51229381
0	3	7.0	1890.0	0.0	2003.0	0.0	98038	47.36840673

long	sqft_living15	sqft_lot15
-122.25677536	1340.0	5650.0
-122.3188624	1690.0	7639.0
-122.23319601	2720.0	8062.0
-122.39318505	1360.0	5000.0
-122.04490059	1800.0	7503.0
-122.00528655	4760.0	101930.0
-122.32704857	2238.0	6819.0
-122.31457273	1650.0	9711.0
-122.33659507	1780.0	8113.0
-122.0308176	2390.0	7570.0

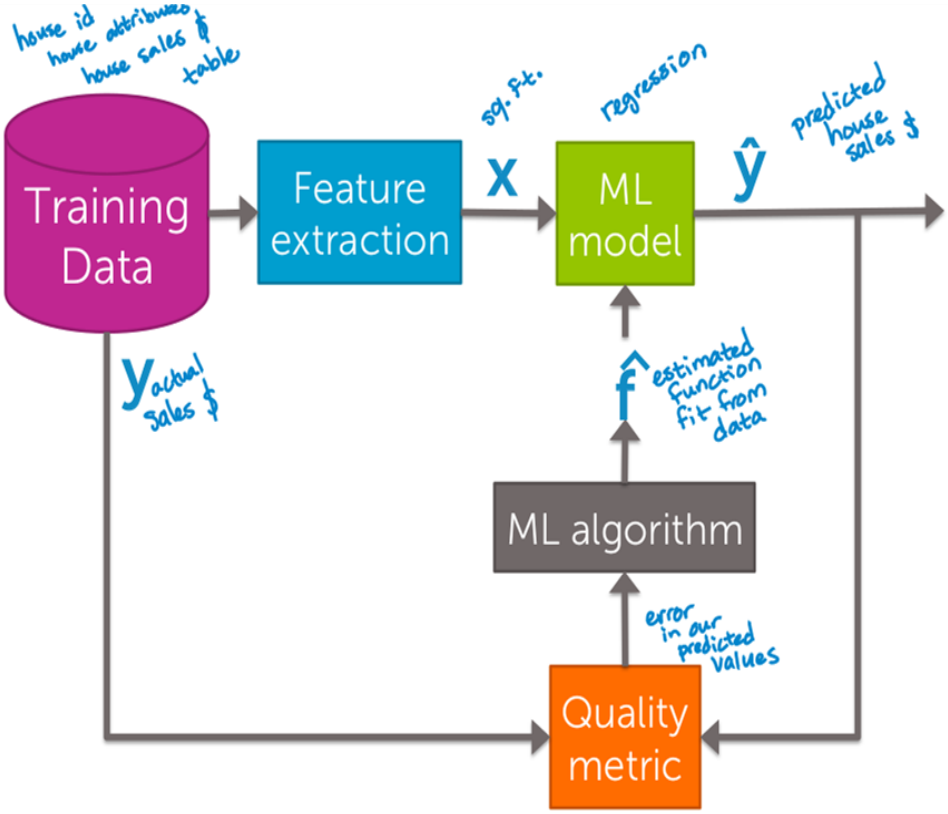
[21613 rows x 21 columns]

Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

```
# Compute RSS when using squarefeet on TEST data:
rss_prices_on_sqft = 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))
```

```
The RSS of predicting Prices based on Square Feet is : 275402936247141.25
```

Fig.2



Regression Week 2: Multiple Regression (Interpretation)

The goal of this first notebook is to explore multiple regression and feature engineering with existing Turi Create functions.
In this notebook you will use data on house sales in King County to predict prices using multiple regression. You will:

Use SFrames to do some feature engineering

Use built-in Turi Create functions to compute the regression weights (coefficients/parameters)

Given the regression weights, predictors and outcome write a function to compute the Residual Sum of Squares

Look at coefficients and interpret their meanings

Evaluate multiple models via RSS

Generic basis expansion ;

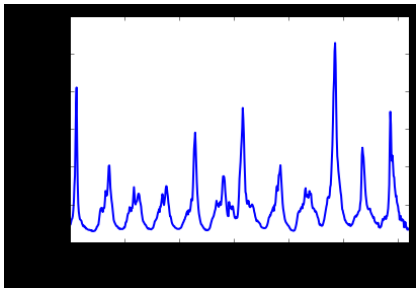
Model: $y_i = w_0 h_0(x_i) + w_1 h_1(x_i) + \ldots + w_D h_D(x_i) + \varepsilon_i = \sum_{j=0}^{D} w_j h_j(x_i) + \varepsilon_i$

feature 1 = h0(x)…often 1 (constant)
feature 2 = h1(x)… e.g., x

feature 3 = h2(x)… e.g., x2 or sin(2πx/12)
.
.
.
feature D+1 = hD(x)… e.g., xp

examples of seasonality where multiple regression is used:

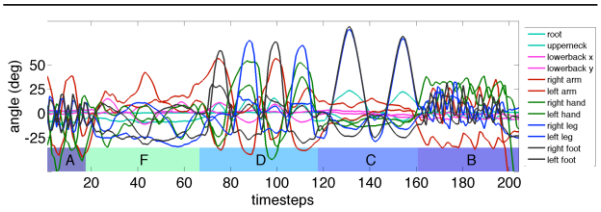
Flu monitoring



Demand forecasting

e.g., jacket purchases

Motion capture data



Weather modeling

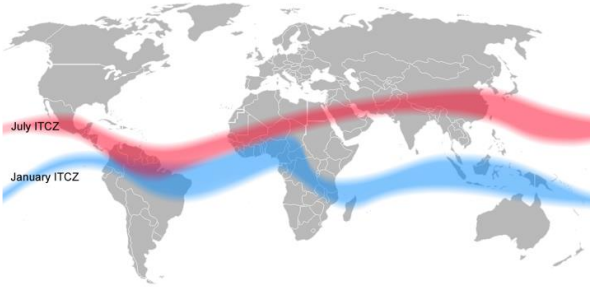


Fig.3

```
import turicreate

sales = turicreate.SFrame('home_data.sframe')
import numpy as np

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 along with the others:
    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_sframe (now including constant):
    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)
```

```
(example_features, example_output) = get_numpy_data(sales, ['sqft_living'], 'price') # the [] around 'sqft_living' makes it a list
print ((example_features[0,:])) # this accesses the first row of the data the ':' indicates 'all columns'
print ((example_output[0])) # and the corresponding output
```

```
[1.00e+00 1.18e+03]
221900.0
```

```
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
```

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

    predictions = np.dot(feature_matrix,weights)
    return(predictions)
```

```
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
```

```
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)
```

```
(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 ":" indicates "all rows"
derivative = feature_derivative(errors, feature)
print ((derivative))
print ((-np.sum(example_output)*2)) # should be the same as derivative
-23345850022.0
-23345850022.0


from math import sqrt # recall that the magnitude/length of a vector [g[0], g[1], g[2]] is sqrt(g[0]^2 + g[1]^2 + g[2]^2)

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]:
            feature = feature_matrix[:,i]
            derivative = feature_derivative(errors,feature)
            # add the squared value of the derivative to the gradient sum of squares (for assessing convergence)
            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 magnitude:
        gradient_magnitude = sqrt(gradient_sum_squares)
        if gradient_magnitude < tolerance:
            converged = True
    return(weights)

# 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]:
#             feature = feature_matrix[:, i]
#             derivative = feature_derivative(errors, feature)
```

```
#           # add the squared value of the derivative to the gradient sum of squares (for assessing convergence)
#           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 matnigude:
#       gradient_magnitude = sqrt(gradient_sum_squares)
#       if gradient_magnitude < tolerance:
#           converged = True
#    return(weights)
train_data,test_data = sales.random_split(.8,seed=0)

# 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

my_sqft_weights = regression_gradient_descent(simple_feature_matrix, output, initial_weights, step_size,
tolerance)
print( (my_sqft_weights))
```

```
[-46999.88716555    281.91211912]
```

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

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

test_predictions = predict_output(test_simple_feature_matrix, my_sqft_weights)
print ((test_predictions))
```

```
[356134.44317093 784640.86422788 435069.83652353 ... 663418.65300782
 604217.10799338 240550.4743332 ]
```

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

```
print ((np.rint(test_predictions[0])))
```

```
356134.0
```

```
test_errors = test_predictions - test_output
RSS = (test_errors*test_errors).sum()
print (RSS)
```

```
275400047593155.94
```

2.1 Running a multiple regression

```
model_features = ['sqft_living', 'sqft_living15'] # sqft_living15 is the average squarefeet for the nearest 15 neighbors.
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
```

```
model_weights = regression_gradient_descent(feature_matrix, output, initial_weights, step_size, tolerance)
print (model_weights)
```

```
[-9.99999688e+04  2.45072603e+02  6.52795277e+01]
```

```
(test_feature_matrix,test_output) = get_numpy_data(test_data,model_features,my_output)
new_test_predictions = predict_output(test_feature_matrix,model_weights)
```

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

```
print (np.rint(new_test_predictions[0]))
```

```
366651.0
```

```
print (test_data[0]['price'])
```

```
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?¶

```
test_errors = new_test_predictions - test_output
RSS = (test_errors*test_errors).sum()
print (RSS)
```

```
270263446465244.06
```

Regression Week 3: Assessing Fit (polynomial regression)

In this notebook you will compare different regression models in order to assess which model fits best. We will be using polynomial regression as a means to examine this topic. In particular you will:

Write a function to take an SArray and a degree and return an SFrame where each column is the SArray to a polynomial value up to the total degree e.g. degree = 3 then column 1 is the SArray column 2 is the SArray squared and column 3 is the SArray cubed

Use matplotlib to visualize polynomial regressions

Use matplotlib to visualize the same polynomial degree on different subsets of the data

Use a validation set to select a polynomial degree

Assess the final fit using test data

We will continue to use the House data from previous notebooks.

import turicreate

```
tmp = turicreate.SArray([1., 2., 3.])
tmp_cubed = tmp.apply(lambda x: x**3)
print (tmp)
print (tmp_cubed)
```

```
[1.0, 2.0, 3.0]
[1.0, 8.0, 27.0]
```

```
ex_sframe = turicreate.SFrame()
ex_sframe['power_1'] = tmp
print( ex_sframe )
```

```
+---------+
| power_1 |
+---------+
|   1.0   |
|   2.0   |
|   3.0   |
+---------+
[3 rows x 1 columns]
```

```
def polynomial_sframe(feature, degree):
    # assume that degree >= 1
    # initialize the SFrame:
    poly_sframe = turicreate.SFrame()
    # and set poly_sframe['power_1'] equal to the passed feature
    poly_sframe['power_1'] = feature
    # first check if degree > 1
    if degree > 1:
        # then loop over the remaining degrees:
```

```
    # range usually starts at 0 and stops at the endpoint-1. We want it to start at 2 and stop at degree
    for power in range(2, degree+1):
        # first we'll give the column a name:
        name = 'power_' + str(power)
        # then assign poly_sframe[name] to the appropriate power of feature
        poly_sframe[name] = feature.apply(lambda x: x**power)
  return poly_sframe

print (polynomial_sframe(tmp, 4))
+---------+---------+---------+---------+
| power_1 | power_2 | power_3 | power_4 |
+---------+---------+---------+---------+
|  1.0    |  1.0    |  1.0    |  1.0    |
|  2.0    |  4.0    |  8.0    |  16.0   |
|  3.0    |  9.0    |  27.0   |  81.0   |
+---------+---------+---------+---------+
[3 rows x 4 columns]

sales = turicreate.SFrame('home_data.sframe')

sales = sales.sort(['sqft_living', 'price'])

poly1_data = polynomial_sframe(sales['sqft_living'], 1)
poly1_data['price'] = sales['price'] # add price to the data since it's the target

model1 = turicreate.linear_regression.create(poly1_data, target = 'price', features = ['power_1'], validation_set =
None)
#let's take a look at the weights before we plot
model1.coefficients
```

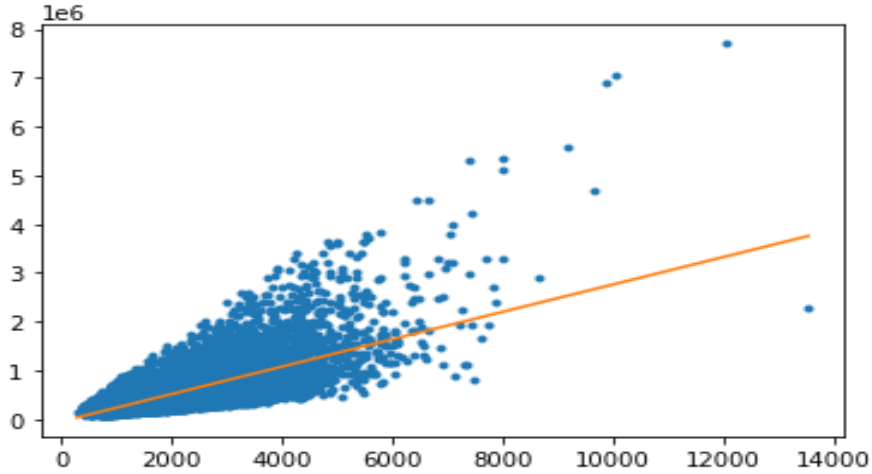
name	index	value	stderr
(intercept)	None	-43579.08525145298	4402.689697427734
power_1	None	280.62277088584864	1.9363985551321306

[2 rows x 4 columns]

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(poly1_data['power_1'],poly1_data['price'],'.',
     poly1_data['power_1'], model1.predict(poly1_data),'-')
[<matplotlib.lines.Line2D at 0x7fcf91fc5890>,
 <matplotlib.lines.Line2D at 0x7fcf90d0f250>]
```

Fig.4




```
poly2_data = polynomial_sframe(sales['sqft_living'], 2)
my_features = poly2_data.column_names() # get the name of the features
poly2_data['price'] = sales['price'] # add price to the data since it's the target
model2 = turicreate.linear_regression.create(poly2_data, target = 'price', features = my_features, validation_set = None)
```

```
model2.coefficients
```

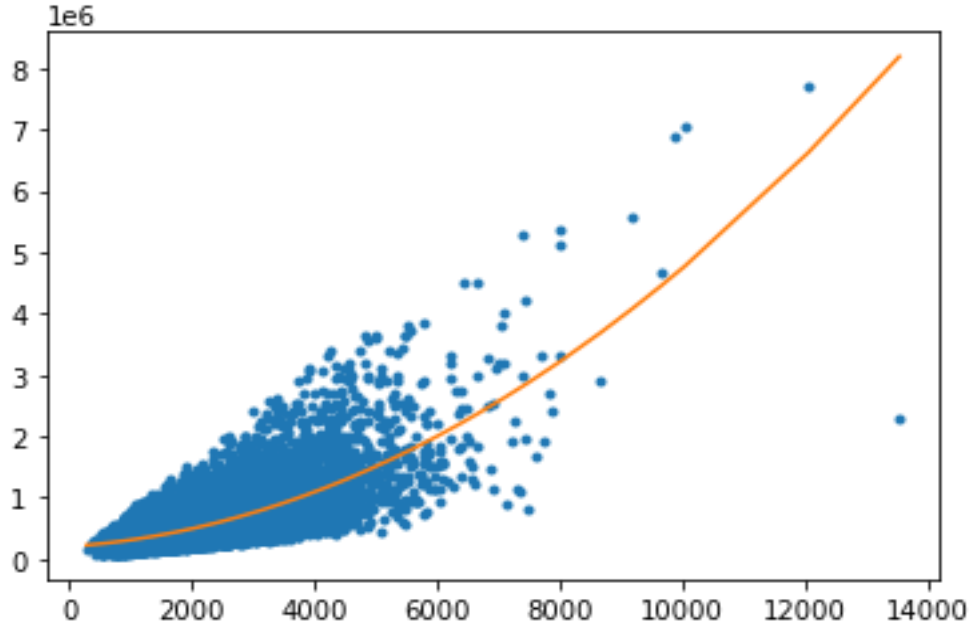
name	index	value	stderr
(intercept)	None	199222.4964446181	7058.004835516453
power_1	None	67.99406406774024	5.28787201316191
power_2	None	0.03858123127891538	0.0008982465470323662

[3 rows x 4 columns]

```
plt.plot(poly2_data['power_1'],poly2_data['price'],'.',
    poly2_data['power_1'], model2.predict(poly2_data),'-')
```

[<matplotlib.lines.Line2D at 0x7fcf90c42390>,
<matplotlib.lines.Line2D at 0x7fcf90c425d0>]

Fig.5



```
poly15_data = polynomial_sframe(sales['sqft_living'], 15)
my_features15 = poly15_data.column_names() # get the name of the features
poly15_data['price'] = sales['price'] # add price to the data since it's the target
model15 = turicreate.linear_regression.create(poly15_data, target = 'price', features = my_features15,
validation_set = None)
```

```
model15.coefficients
```

name	index	value	stderr
(intercept)	None	73619.75208492432	425692.5901395389
power_1	None	410.28746259266694	1985.667355590269
power_2	None	-0.23045071447008592	3.873483280788419
power_3	None	7.588405425789273e-05	0.004172386192710933
power_4	None	-5.657018028046504e-09	2.764105548382603e-06
power_5	None	-4.57028130597624e-13	1.1849596849784023e-09
power_6	None	2.6636020653190584e-17	3.353137065632936e-13
power_7	None	3.3858476930298646e-21	6.235317231351369e-17
power_8	None	1.1472310403747302e-25	7.477909973647464e-21
power_9	None	-4.6529358524593984e-30	6.325367370855489e-25

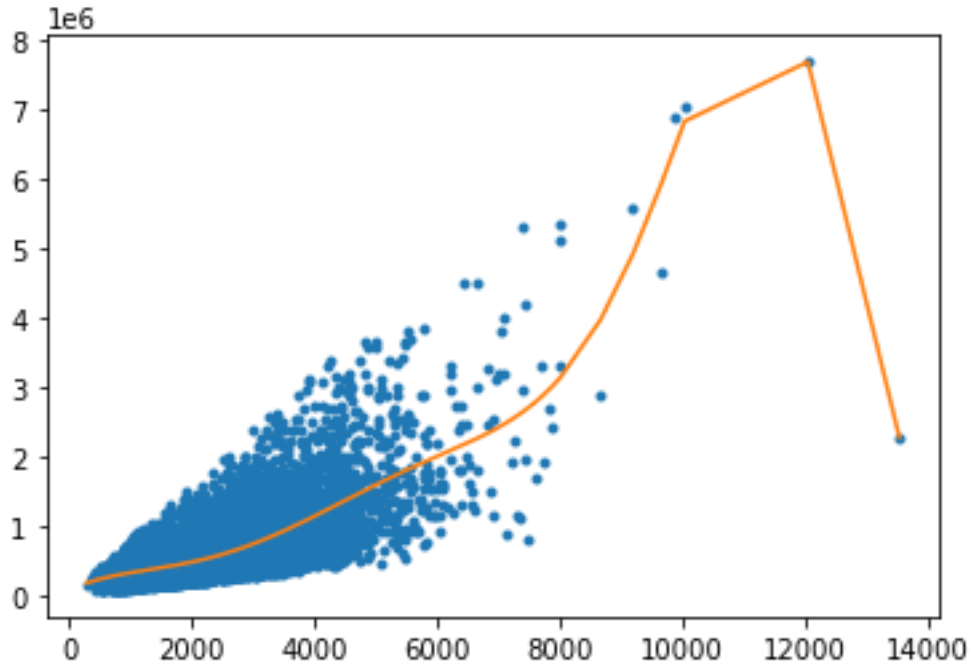
[16 rows x 4 columns]

Note: Only the head of the SFrame is printed.

You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

```
plt.plot(poly15_data['power_1'],poly15_data['price'],'.',
    poly15_data['power_1'], model15.predict(poly15_data),'-')
```

Fig.6



Changing the data and re-learning

We're going to split the sales data into four subsets of roughly equal size. Then you will estimate a 15th degree polynomial model on all four subsets of the data. Print the coefficients (you should use .print_rows(num_rows = 16) to view all of them) and plot the resulting fit (as we did above). The quiz will ask you some questions about these results.
To split the sales data into four subsets, we perform the following steps:
First split sales into 2 subsets with .random_split(0.5, seed=0).
Next split the resulting subsets into 2 more subsets each. Use .random_split(0.5, seed=0).

We set seed=0 in these steps so that different users get consistent results. You should end up with 4 subsets (set_1, set_2, set_3, set_4) of approximately equal size.

```
set_11, set_22 = sales.random_split(0.5, seed=0)
set_1, set_2 = set_11.random_split(0.5, seed=0)
set_3, set_4 = set_22.random_split(0.5, seed=0)
```

```
set_1_data = polynomial_sframe(set_1['sqft_living'], 15)
my_features_1 = set_1_data.column_names() # get the name of the features
set_1_data['price'] = set_1['price'] # add price to the data since it's the target
model_1 = turicreate.linear_regression.create(set_1_data, target = 'price', features = my_features_1,
validation_set = None)
model_1.coefficients.print_rows(num_rows = 16)
plt.plot(set_1_data['power_1'],set_1_data['price'],'.',set_1_data['power_1'], model_1.predict(set_1_data),'-')
```

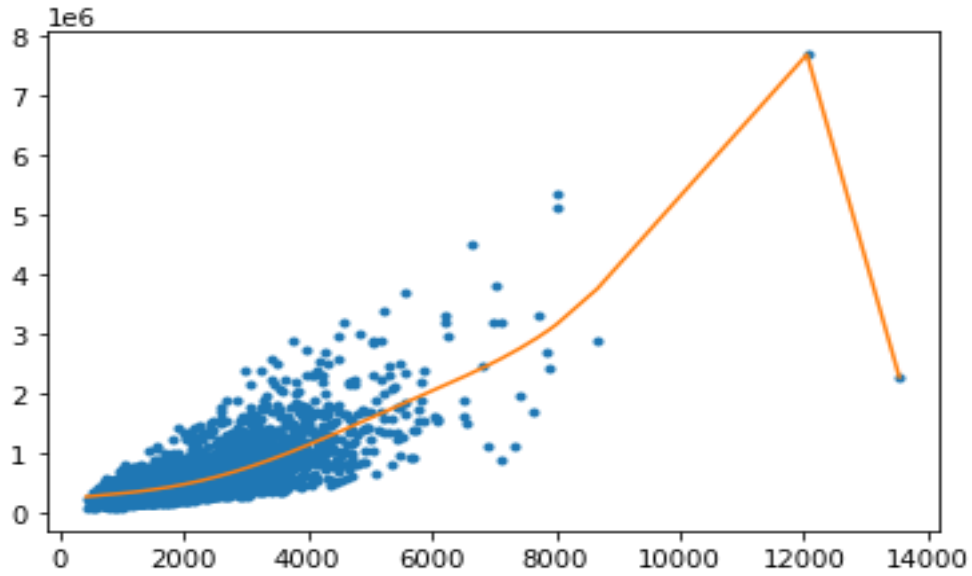


Fig.7

```
set_2_data = polynomial_sframe(set_2['sqft_living'], 15)
my_features_2 = set_2_data.column_names() # get the name of the features
set_2_data['price'] = set_2['price'] # add price to the data since it's the target
model_2 = turicreate.linear_regression.create(set_1_data, target = 'price', features = my_features_2,
validation_set = None)
model_2.coefficients.print_rows(num_rows = 16)
plt.plot(set_2_data['power_1'],set_2_data['price'],'.',set_2_data['power_1'], model_2.predict(set_2_data),'-')
```

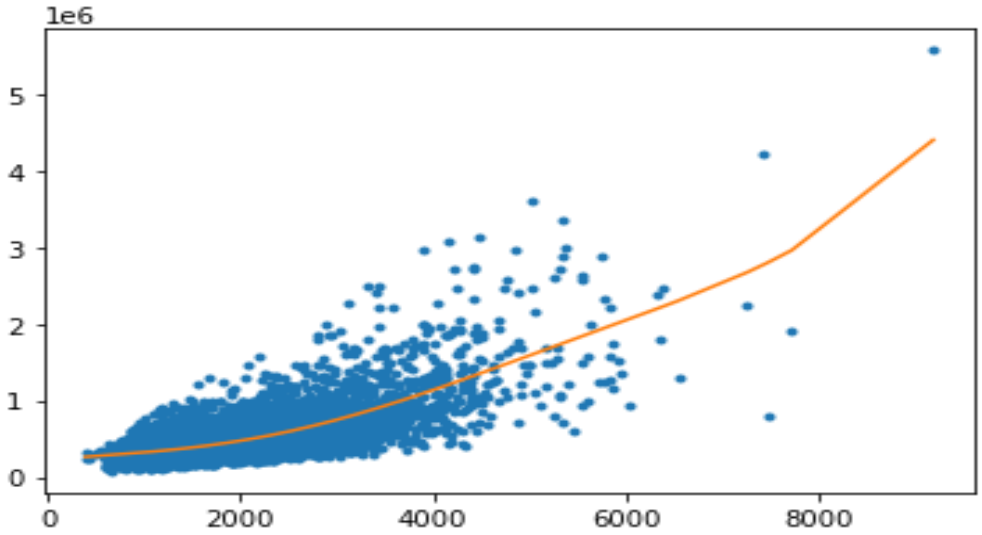


Fig.8

```
set_3_data = polynomial_sframe(set_3['sqft_living'], 15)
my_features_3 = set_3_data.column_names() # get the name of the features
set_3_data['price'] = set_3['price'] # add price to the data since it's the target

model_3 = turicreate.linear_regression.create(set_3_data, target = 'price', features = my_features_3,
validation_set = None)
model_3.coefficients.print_rows(num_rows = 16)
plt.plot(set_3_data['power_1'],set_3_data['price'],'.',set_3_data['power_3'], model_3.predict(set_3_data),'-')
```

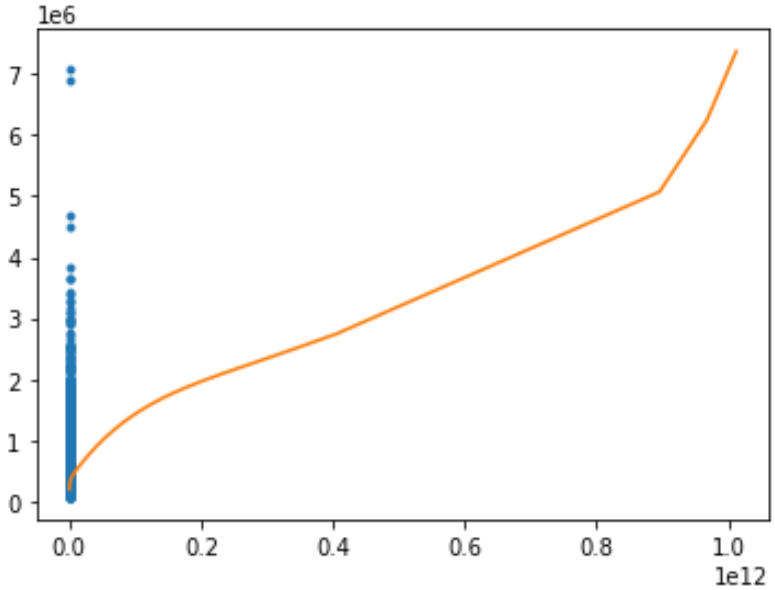


Fig.9

```
set_4_data = polynomial_sframe(set_4['sqft_living'], 15)
my_features_4 = set_4_data.column_names() # get the name of the features
set_4_data['price'] = set_4['price'] # add price to the data since it's the target

model_4 = turicreate.linear_regression.create(set_4_data, target = 'price', features = my_features_4,
validation_set = None)
model_4.coefficients.print_rows(num_rows = 16)
```

```
plt.plot(set_4_data['power_4'],set_4_data['price'],'.',set_4_data['power_4'], model_4.predict(set_4_data),'-')
```

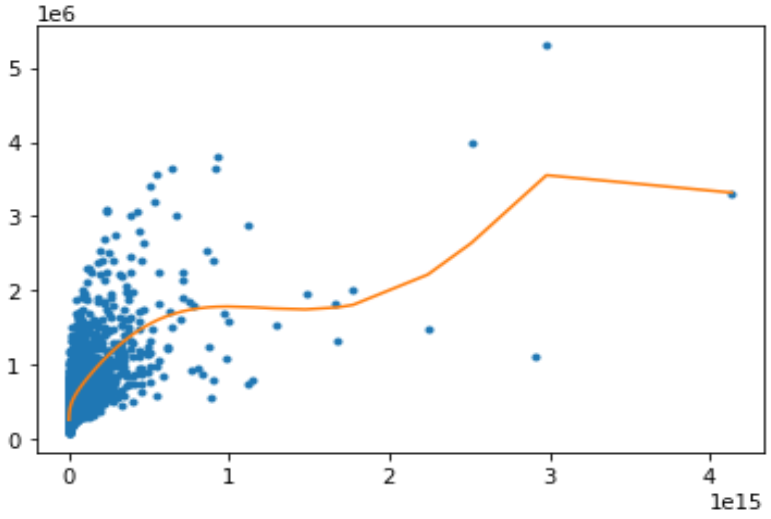


Fig..10

Regression Week 4: Ridge Regression (interpretation)

In this notebook, we will run ridge regression multiple times with different L2 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 L2 regularization. In particular, we will:
Use a pre-built implementation of regression (Turi Create) to run polynomial regression
Use matplotlib to visualize polynomial regressions
Use a pre-built implementation of regression (Turi Create) to run polynomial regression, this time with L2 penalty¶
Use matplotlib to visualize polynomial regressions under L2 regularization
Choose best L2 penalty using cross-validation.
Assess the final fit using test data.
We will continue to use the House data from previous notebooks. (In the next programming assignment for this module, you will implement your own ridge regression learning algorithm using gradient descent.)

```
import turicreate
```

4.1 Polynomial regression, revisited

We build on the material from Week 3, where we wrote the function to produce an SFrame with columns containing the powers of a given input. Copy and paste the function polynomial_sframe from Week 3:

```
def polynomial_sframe(feature, degree):
  # assume that degree >= 1
  # initialize the SFrame:
  poly_sframe = turicreate.SFrame()
  # and set poly_sframe['power_1'] equal to the passed feature
  poly_sframe['power_1'] = feature
  # first check if degree > 1
  if degree > 1:
    # then loop over the remaining degrees:
```

```
    # range usually starts at 0 and stops at the endpoint-1. We want it to start at 2 and stop at degree
    for power in range(2, degree+1):
        # first we'll give the column a name:
        name = 'power_' + str(power)
        # then assign poly_sframe[name] to the appropriate power of feature
        tmp = feature.apply(lambda x: x**power)
        poly_sframe[name] = tmp
  return poly_sframe

import matplotlib.pyplot as plt
%matplotlib inline

sales = turicreate.SFrame('home_data.sframe')
```

As in Week 3, we will use the sqft_living variable. For plotting purposes (connecting the dots), you'll need to sort by the values of sqft_living. For houses with identical square footage, we break the tie by their prices.

```
sales = sales.sort(['sqft_living','price'])

l2_small_penalty = 1e-5
l2_penalty=1e5
```

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=1e-5) to make the solution numerically stable. (In lecture, we discussed the fact that regularization can also help with numerical stability, and here we are seeing a practical example.)
With the L2 penalty specified above, fit the model and print out the learned weights.
Hint: make sure to add 'price' column to the new SFrame before calling turicreate.linear_regression.create(). Also, make sure Turi Create doesn't create its own validation set by using the option validation_set=None in this call.

```
poly15_data = polynomial_sframe(sales['sqft_living'],15)
my_features = poly15_data.column_names()
poly15_data['price'] = sales['price']
model15 =
turicreate.linear_regression.create(poly15_data,target='price',features=my_features,l2_penalty=l2_small_penalt
y,validation_set=None)

model15.coefficients
```

name	index	value	stderr
(intercept)	None	167924.86687009025	425633.59551983344
power_1	None	103.09092101866923	1985.392171823349
power_2	None	0.1346045889804318	3.872946473997841
power_3	None	-0.0001290713872668172	0.00417180796245181
power_4	None	5.189290320618038e-08	2.7637224847366063e-06
power_5	None	-7.771694203237427e-12	1.1847954673067752e-09
power_6	None	1.7114488922927697e-16	3.352672371037377e-13
power_7	None	4.5117806197111717e-20	6.234453109735613e-17
power_8	None	-4.788409152736236e-25	7.476873647281194e-21

```
 power_9    None   -2.333434766152199e-28    6.324490769638591e-25
```
[16 rows x 4 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

QUIZ QUESTION: What's the learned value for the coefficient of feature power_1?
Ans 103

```
(semi_split1, semi_split2) = sales.random_split(.5,seed=0)
(set_1, set_2) = semi_split1.random_split(0.5, seed=0)
(set_3, set_4) = semi_split2.random_split(0.5, seed=0)
```
Next, fit a 15th degree polynomial on set_1, set_2, set_3, and set_4, using 'sqft_living' to predict prices. Print the weights and make a plot of the resulting model.
Hint: When calling turicreate.linear_regression.create(), use the same L2 penalty as before (i.e. l2_small_penalty). Also, make sure Turi Create doesn't create its own validation set by using the option validation_set = None in this call.

```
poly01_data = polynomial_sframe(set_1['sqft_living'],15)
my_features = poly01_data.column_names()#get the names of features
poly01_data['price'] = set_1['price']#add price to the data since it's the target
model01 = turicreate.linear_regression.create(poly01_data, target = 'price',features = my_features,l2_penalty =
l2_small_penalty,validation_set = None)
model01.coefficients.print_rows(num_rows=16)
plt.plot(poly01_data['power_1'],poly01_data['price'],'.',
     poly01_data['power_1'],model01.predict(poly01_data),'-')
```

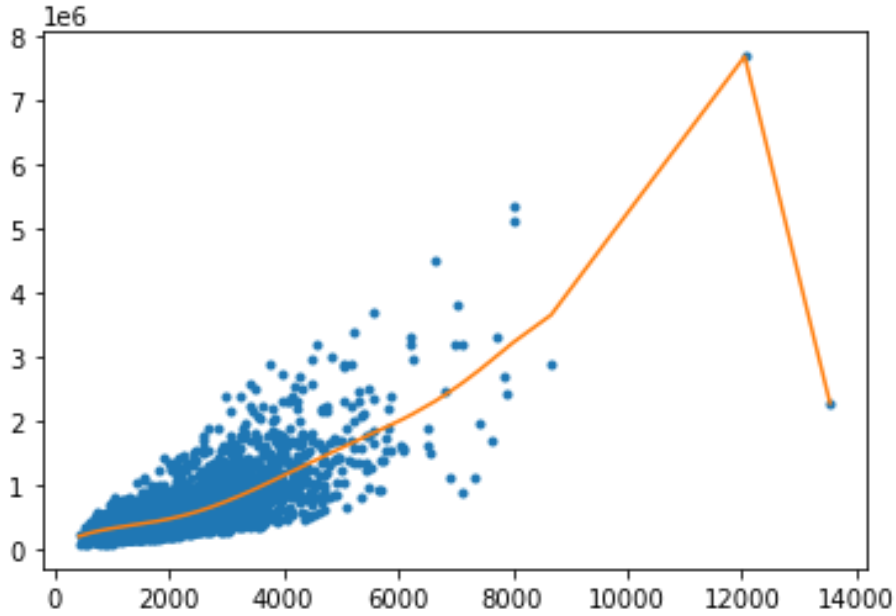


Fig.11

```
poly02_data = polynomial_sframe(set_2['sqft_living'],15)
my_features = poly02_data.column_names() #this is to get the name of features
poly02_data['price'] = set_2['price']
model02 = turicreate.linear_regression.create(poly02_data,target = 'price',features = my_features,l2_penalty =
l2_small_penalty,validation_set = None)
model02.coefficients.print_rows(num_rows = 16)
plt.plot(poly02_data['power_1'],poly02_data['price'],'.',
    poly02_data['power_1'],model02.predict(poly02_data),'-')
```

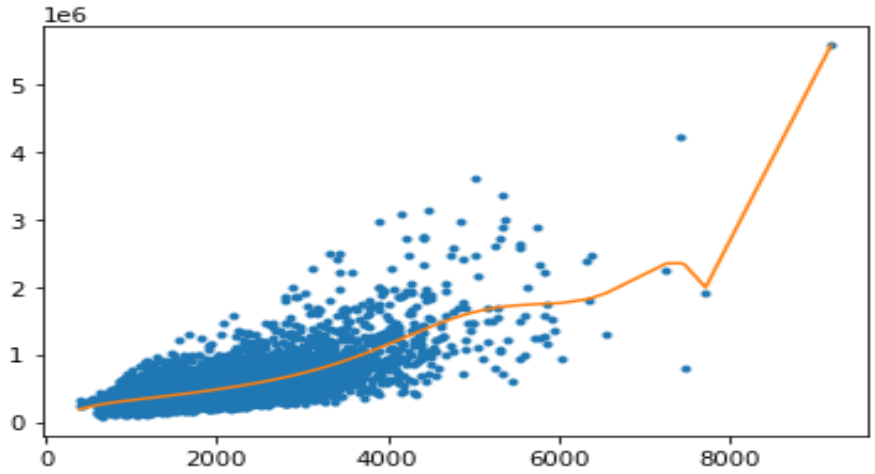


Fig.12

```
poly04_data = polynomial_sframe(set_4['sqft_living'],15)
my_features = poly04_data.column_names()#to get name of features
poly04_data['price'] = set_4['price']
model04 = turicreate.linear_regression.create(poly04_data,features = my_features,target='price',l2_penalty =
l2_small_penalty,validation_set = None)
model04.coefficients.print_rows(num_rows=16)
plt.plot(poly04_data['power_1'],poly04_data['price'],'.',
    poly04_data['power_1'],model04.predict(poly04_data),'-')
```

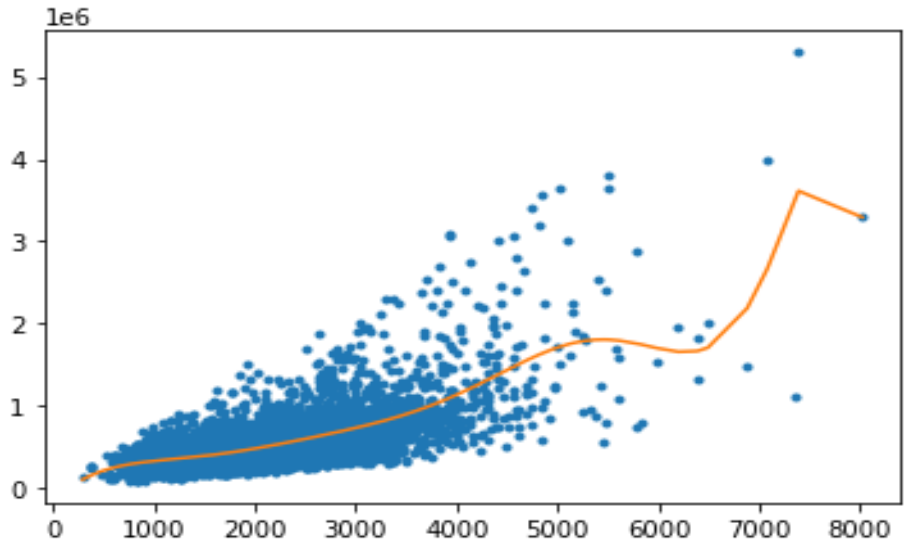


Fig.13

Ridge regression comes to rescue

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. (Weights of model15 looked quite small, but they are not that small because 'sqft_living' input is in the order of thousands.)

With the argument l2_penalty=1e5, fit a 15th-order polynomial model on set_1, set_2, set_3, and set_4. Other than the change in the l2_penalty parameter, the code should be the same as the experiment above. Also, make sure Turi Create doesn't create its own validation set by using the option validation_set = None in this call.

```
poly01_data = polynomial_sframe(set_1['sqft_living'], 15)
my_features = poly01_data.column_names() # get the name of the features
poly01_data['price'] = set_1['price'] # add price to the data since it's the target
model01 = turicreate.linear_regression.create(poly01_data, target = 'price', features = my_features,
l2_penalty=l2_penalty,validation_set = None)
model01.coefficients.print_rows(num_rows = 16)
plt.plot(poly01_data['power_1'],poly01_data['price'],'.',
    poly01_data['power_1'], model01.predict(poly01_data),'-')
```

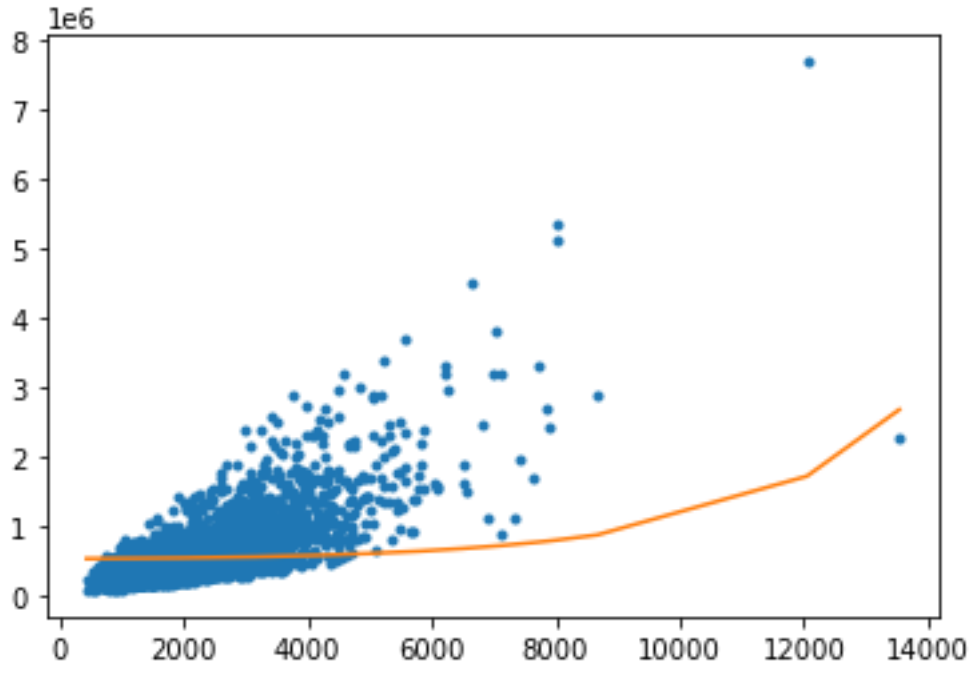


Fig.14

```
poly02_data = polynomial_sframe(set_2['sqft_living'], 15)
my_features = poly02_data.column_names() # get the name of the features
poly02_data['price'] = set_2['price'] # add price to the data since it's the target
model02 = turicreate.linear_regression.create(poly02_data, target = 'price', features =
my_features,l2_penalty=l2_penalty, validation_set = None)
model02.coefficients.print_rows(num_rows = 16)
plt.plot(poly02_data['power_1'],poly02_data['price'],'.',
    poly02_data['power_1'], model02.predict(poly02_data),'-')
```

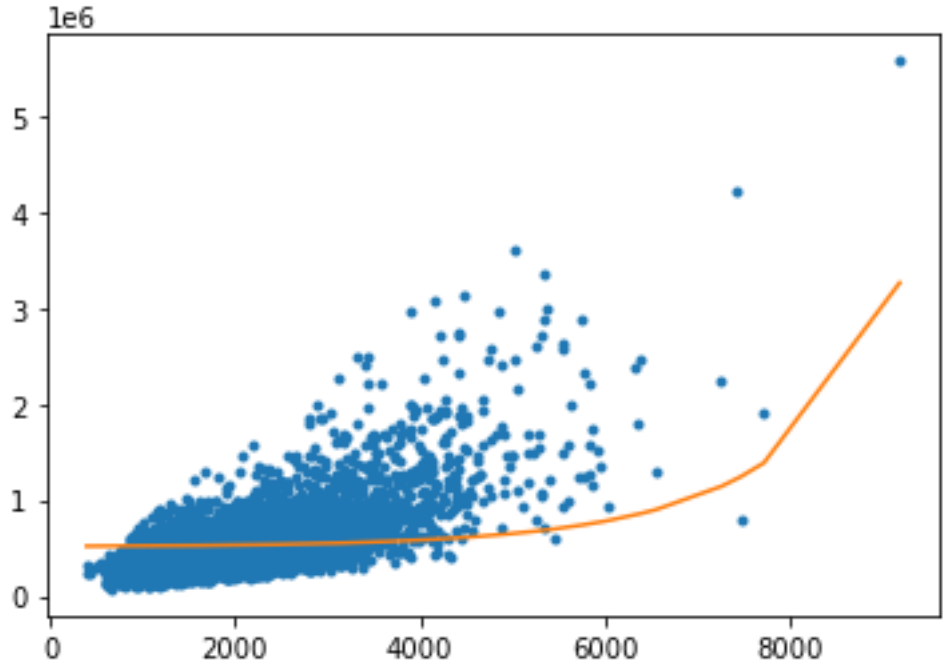


Fig.15

```
poly04_data = polynomial_sframe(set_4['sqft_living'], 15)
my_features = poly04_data.column_names() # get the name of the features
poly04_data['price'] = set_4['price'] # add price to the data since it's the target
model04 = turicreate.linear_regression.create(poly04_data, target = 'price', features =
my_features,l2_penalty=l2_penalty, validation_set = None)
model04.coefficients.print_rows(num_rows = 16)
plt.plot(poly04_data['power_1'],poly04_data['price'],'.',
     poly04_data['power_1'], model04.predict(poly04_data),'-')
```

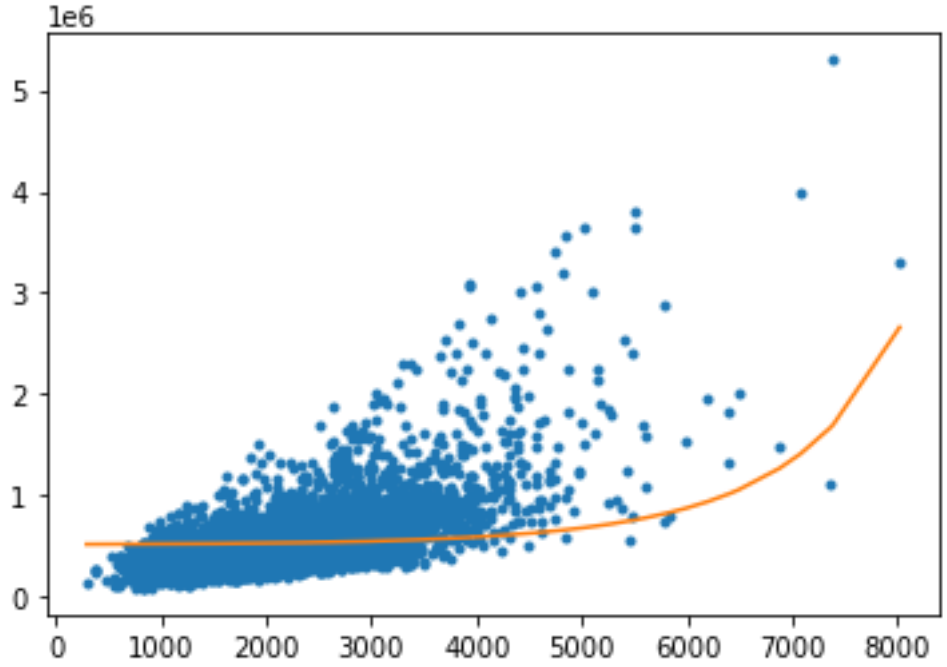


Fig.16

4.2 Selecting an L2 penalty via cross-validation

Just like the polynomial degree, the L2 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 roughtly 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:

Set aside segment 0 as the validation set, and fit a model on rest of data, and evalutate it on this validation set Set aside segment 1 as the validation set, and fit a model on rest of data, and evalutate it on this validation set ... Set aside segment k-1 as the validation set, and fit a model on rest of data, and evalutate it on this validation set

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. The package turicreate_cross_validation (see below) has a utility function for shuffling a given SFrame. We reserve 10% of the data as the test set and shuffle the remainder. (Make sure to use seed=1 to get consistent answer.)

```
import turicreate_cross_validation.cross_validation as tcv

(train_valid, test) = sales.random_split(.9, seed=1)
train_valid_shuffled = tcv.shuffle_sframe(train_valid, random_seed=1)
```

Once the data is shuffled, we divide it 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 range(k):
    start = (n*i)/k
    end = (n*(i+1))/k-1
    print(i, (start, end))
```

```
0 (0.0, 1938.6)
1 (1939.6, 3878.2)
2 (3879.2, 5817.8)
3 (5818.8, 7757.4)
4 (7758.4, 9697.0)
5 (9698.0, 11636.6)
6 (11637.6, 13576.2)
7 (13577.2, 15515.8)
```

8 (15516.8, 17455.4)
9 (17456.4, 19395.0)

Let us familiarize ourselves with array slicing with SFrame. To extract a continuous slice from an SFrame, 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] # rows 0 to 9
```

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
8645511350	2014-12-01 00:00:00+00:00	300000.0	3.0	1.75	1810.0	21138.0	1.0	0
7237501370	2014-07-17 00:00:00+00:00	1079000.0	4.0	3.25	4800.0	12727.0	2.0	0
7278700100	2015-01-21 00:00:00+00:00	625000.0	4.0	2.5	2740.0	9599.0	1.0	0
1421079007	2015-03-24 00:00:00+00:00	408506.0	3.0	2.75	2480.0	209199.0	1.5	0
4338800370	2014-11-17 00:00:00+00:00	220000.0	3.0	1.0	1000.0	6020.0	1.0	0
7511200020	2014-08-29 00:00:00+00:00	509900.0	3.0	1.75	1690.0	53578.0	1.0	0
3300701615	2014-09-30 00:00:00+00:00	655000.0	4.0	2.5	2630.0	4000.0	3.0	0
7011200260	2014-12-19 00:00:00+00:00	485000.0	4.0	2.0	1400.0	3600.0	1.0	0
3570000130	2014-06-11 00:00:00+00:00	580379.0	4.0	2.75	2240.0	27820.0	1.5	0
2796100640	2015-04-24 00:00:00+00:00	264900.0	4.0	2.5	2040.0	7000.0	1.0	0

view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat
0	4	7.0	1240.0	570.0	1977.0	0.0	98058	47.46736904
0	3	10.0	4800.0	0.0	2011.0	0.0	98059	47.53108576
2	3	8.0	1820.0	920.0	1961.0	0.0	98177	47.77279701
0	3	8.0	1870.0	610.0	2000.0	0.0	98010	47.30847072
0	3	6.0	1000.0	0.0	1944.0	0.0	98166	47.47933643
0	3	8.0	1690.0	0.0	1984.0	0.0	98053	47.6545751
0	3	8.0	2630.0	0.0	2002.0	0.0	98117	47.69151411
0	3	7.0	1100.0	300.0	1900.0	0.0	98119	47.63846783
0	4	8.0	2240.0	0.0	1976.0	0.0	98075	47.59357299
0	3	7.0	1250.0	790.0	1979.0	0.0	98031	47.40555074

long	sqft_living15	sqft_lot15
-122.17768631	1850.0	12200.0
-122.13389261	4750.0	13602.0
-122.38485302	2660.0	8280.0
-121.88816296	2040.0	219229.0
-122.34575463	1300.0	8640.0
-122.04899568	2290.0	52707.0
-122.38139901	1640.0	4000.0
-122.36993806	1630.0	2048.0
-122.05362447	2330.0	20000.0
-122.17648783	1900.0	7378.0

[10 rows x 21 columns]

Now let us extract individual segments with array slicing. Consider the scenario where we group the houses in the train_valid_shuffled dataframe into k=10 segments of roughly equal size, with starting and ending indices computed as above. Extract the fourth segment (segment 3) and assign it to a variable called validation4.

```
validation4 = train_valid_shuffled[5818.8:7757.4]

print (int(round(validation4['price'].mean()
           , 0)))
```

```
559642
```

After designating one of the k segments as the validation set, we train a model using the rest of the data. To choose the remainder, we slice (0:start) and (end+1:n) of the data and paste them together. SFrame has append() method that pastes together two disjoint sets of rows originating from a common dataset. For instance, the following cell pastes together the first and last two rows of the train_valid_shuffled dataframe.

```
n = len(train_valid_shuffled)
first_two = train_valid_shuffled[0:2]
last_two = train_valid_shuffled[n-2:n]
print (first_two.append(last_two))

train4 = train_valid_shuffled[0:5818].append(train_valid_shuffled[7757:n])

print (int(round(train4['price'].mean(), 0)))
```

```
536850
```

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, (iv) name of output column (e.g. price) and (v) list of feature names. The function returns the average validation error using k segments as validation sets.
For each i in [0, 1, ..., k-1]: Compute starting and ending indices of segment i and call 'start' and 'end' Form validation set by taking a slice (start:end+1) from the data. Form training set by appending slice (end+1:n) to the end of slice (0:start). Train a linear model using training set just formed, with a given l2_penalty Compute validation error using validation set just formed

```
import numpy as np
def k_fold_cross_validation(k, l2_penalty, data, output_name, features_list):
    empty_vector = np.empty(k)
    n=len(data)
    for i in range(k):
        start = (n*i)/k
```

```
    end = (n*(i+1))/k-1
#    print(i, (start, end))
    validation_set = data[start:end+1]
    train_set = data[0:start].append(data[end+1:n])
    model = turicreate.linear_regression.create(train_set,target=output_name,
features=features_list,l2_penalty=l2_penalty,validation_set=None)

    predict = model.predict(validation_set)
    errors = validation_set[output_name] - predict
    square_errors = errors**2
    RSS = square_errors.sum()
    empty_vector[i] = RSS
  return empty_vector.mean()
  print ('mean: ' + str(empty_vector.mean()))
```

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: We will again be aiming to fit a 15th-order polynomial model using the sqft_living input For l2_penalty in [10^1, 10^1.5, 10^2, 10^2.5, ..., 10^7] (to get this in Python, you can use this Numpy function: np.logspace(1, 7, num=13).) Run 10-fold cross-validation with l2_penalty Report which L2 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!

```
poly_data = polynomial_sframe(train_valid_shuffled['sqft_living'], 15)
my_features = poly_data.column_names() # get the name of the features
poly_data['price'] = train_valid_shuffled['price'] # add price to the data since it's the target
a = np.logspace(1, 7, num=13)
nn = len(a)
error_vector = np.empty(13)
for i in range(nn):
  #print 'l2_penalty: ' + str(l2_penalty)

  error_vector[i] = k_fold_cross_validation(10, a[i], poly_data, 'price', my_features)

  #print 'error_vector: ' + str(error_vector)
print (error_vector)
```

```
Linear regression:
--------------------------------------------------------
Number of examples          : 17457
Number of features          : 15
Number of unpacked features : 15
Number of coefficients    : 16
Starting Newton Method
```

```
# Plot the l2_penalty values in the x axis and the cross-validation error in the y axis.
# Using plt.xscale('log') will make your plot more intuitive.
plt.plot(a,error_vector,'k-')
plt.xlabel('$\ell_2$ penalty')
plt.ylabel('cross validation error')
plt.xscale('log')
plt.yscale('log')
print (a)
```

```
print (error_vector)
```

```
[1.00000000e+01 3.16227766e+01 1.00000000e+02 3.16227766e+02
 1.00000000e+03 3.16227766e+03 1.00000000e+04 3.16227766e+04
 1.00000000e+05 3.16227766e+05 1.00000000e+06 3.16227766e+06
 1.00000000e+07]
[6.24702191e+14 3.59522656e+14 1.82500743e+14 1.24345199e+14
 1.20963608e+14 1.23921949e+14 1.37124115e+14 1.71719393e+14
 2.29172268e+14 2.52982619e+14 2.58749756e+14 2.62867019e+14
 2.64926582e+14]
```

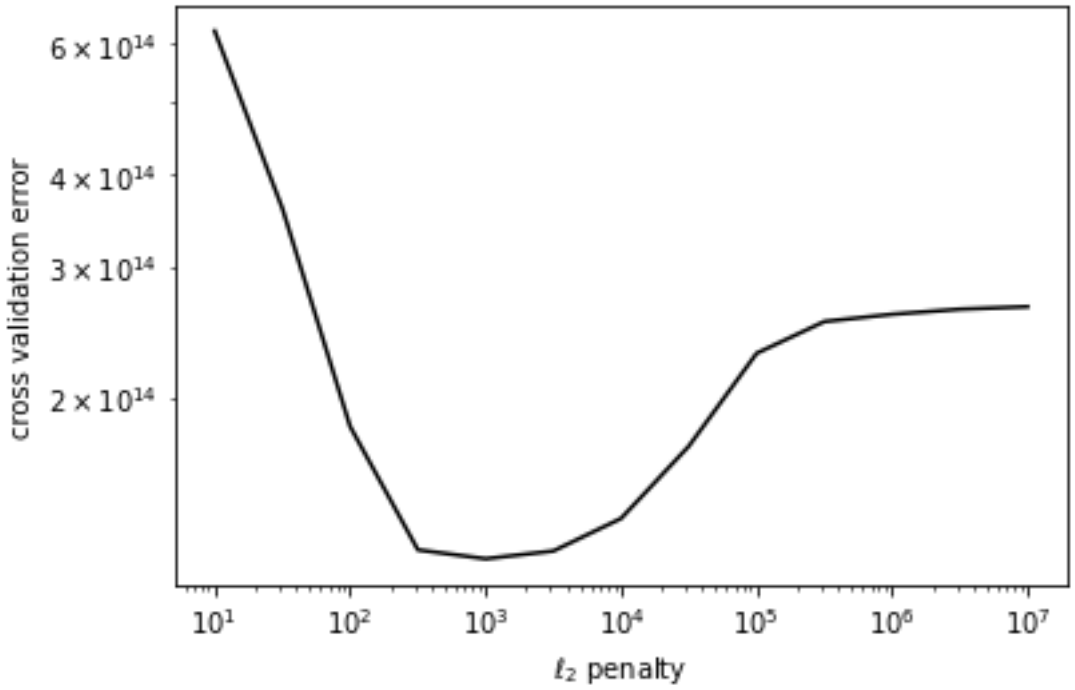


Fig.17

```
l2_penalty = 1e3
data = polynomial_sframe(train_valid['sqft_living'], 15)
my_features = data.column_names() # get the name of the features
data['price'] = train_valid['price'] # add price to the data since it's the target
final_model = turicreate.linear_regression.create(data, target = 'price', features =
my_features,l2_penalty=l2_penalty, validation_set = None)

predict = final_model.predict(test)
errors = test['price'] - predict
square_errors = errors ** 2
RSS = square_errors.sum()
print (RSS)
Linear regression:
--------------------------------------------------------
Number of examples          : 19396
Number of features          : 15
Number of unpacked features : 15
Number of coefficients    : 16
--------------------------------------------------------

=252897427447157.44
```

Regression Week 5: Feature Selection and LASSO(Interpretation)

In this notebook, you will use LASSO to select features, building on a pre-implemented solver for LASSO (using Turi Create, though you can use other solvers). You will:

Run LASSO with different L1 penalties.

Choose best L1 penalty using a validation set.

Choose best L1 penalty using a validation set, with additional constraint on the size of subset.

In the second notebook, you will implement your own LASSO solver, using coordinate descent.

Fire up Turi Create

```
import turicreate
```

Load in house sales data

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

```
sales = turicreate.SFrame('home_data.sframe')
sales
```

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
7129300520	2014-10-13 00:00:00+00:00	221900.0	3.0	1.0	1180.0	5650.0	1.0	0
6414100192	2014-12-09 00:00:00+00:00	538000.0	3.0	2.25	2570.0	7242.0	2.0	0
5631500400	2015-02-25 00:00:00+00:00	180000.0	2.0	1.0	770.0	10000.0	1.0	0
2487200875	2014-12-09 00:00:00+00:00	604000.0	4.0	3.0	1960.0	5000.0	1.0	0
1954400510	2015-02-18 00:00:00+00:00	510000.0	3.0	2.0	1680.0	8080.0	1.0	0
7237550310	2014-05-12 00:00:00+00:00	1225000.0	4.0	4.5	5420.0	101930.0	1.0	0
1321400060	2014-06-27 00:00:00+00:00	257500.0	3.0	2.25	1715.0	6819.0	2.0	0
2008000270	2015-01-15 00:00:00+00:00	291850.0	3.0	1.5	1060.0	9711.0	1.0	0
2414600126	2015-04-15 00:00:00+00:00	229500.0	3.0	1.0	1780.0	7470.0	1.0	0

3793500160	2015-03-12 00:00:00+00:00	323000.0	3.0	2.5	1890.0	6560.0	2.0	0

view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat
0	3	7.0	1180.0	0.0	1955.0	0.0	98178	47.51123398
0	3	7.0	2170.0	400.0	1951.0	1991.0	98125	47.72102274
0	3	6.0	770.0	0.0	1933.0	0.0	98028	47.73792661
0	5	7.0	1050.0	910.0	1965.0	0.0	98136	47.52082
0	3	8.0	1680.0	0.0	1987.0	0.0	98074	47.61681228
0	3	11.0	3890.0	1530.0	2001.0	0.0	98053	47.65611835
0	3	7.0	1715.0	0.0	1995.0	0.0	98003	47.30972002
0	3	7.0	1060.0	0.0	1963.0	0.0	98198	47.40949984
0	3	7.0	1050.0	730.0	1960.0	0.0	98146	47.51229381
0	3	7.0	1890.0	0.0	2003.0	0.0	98038	47.36840673

long	sqft_living15	sqft_lot15
-122.25677536	1340.0	5650.0
-122.3188624	1690.0	7639.0
-122.23319601	2720.0	8062.0
-122.39318505	1360.0	5000.0
-122.04490059	1800.0	7503.0
-122.00528655	4760.0	101930.0
-122.32704857	2238.0	6819.0
-122.31457273	1650.0	9711.0
-122.33659507	1780.0	8113.0
-122.0308176	2390.0	7570.0

[21613 rows x 21 columns]

Note: Only the head of the SFrame is printed.

You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

Create new features

As in Week 2, we consider features that are some transformations of inputs.

```
from math import log, sqrt
sales['sqft_living_sqrt'] = sales['sqft_living'].apply(sqrt)
sales['sqft_lot_sqrt'] = sales['sqft_lot'].apply(sqrt)
sales['bedrooms_square'] = sales['bedrooms']*sales['bedrooms']

# In the dataset, 'floors' was defined with type string,
# so we'll convert them to float, before creating a new feature.
sales['floors'] = sales['floors'].astype(float)
sales['floors_square'] = sales['floors']*sales['floors']
```

Squaring bedrooms will increase the separation between not many bedrooms (e.g. 1) and lots of bedrooms (e.g. 4) since 1^2 = 1 but 4^2 = 16. Consequently this variable will mostly affect houses with many bedrooms.

> On the other hand, taking square root of sqft_living will decrease the separation between big house and small house. The owner may not be exactly twice as happy for getting a house that is twice as big.

5.1 Learn regression weights with L1 penalty

Let us fit a model with all the features available, plus the features we just created above.

```
all_features = ['bedrooms', 'bedrooms_square',
            'bathrooms',
            'sqft_living', 'sqft_living_sqrt',
            'sqft_lot', 'sqft_lot_sqrt',
            'floors', 'floors_square',
            'waterfront', 'view', 'condition', 'grade',
            'sqft_above',
            'sqft_basement',
            'yr_built', 'yr_renovated']
```

Applying L1 penalty requires adding an extra parameter (`l1_penalty`) to the linear regression call in Turi Create. (Other tools may have separate implementations of LASSO.) Note that it's important to set `l2_penalty=0` to ensure we don't introduce an additional L2 penalty.

```
model_all = turicreate.linear_regression.create(sales, target='price', features=all_features,
                                                validation_set=None,
                                                l2_penalty=0., l1_penalty=1e10)
```

Find what features had non-zero weight.

```
# model_all
rows = len(all_features)
model_all.coefficients.print_rows(num_rows=100)
```

```
+------------------+-------+--------------------+--------+
|       name       | index |       value        | stderr |
+------------------+-------+--------------------+--------+
|   (intercept)    |  None | 274873.0559504957  |  None  |
|     bedrooms     |  None |        0.0         |  None  |
|  bedrooms_square |  None |        0.0         |  None  |
|    bathrooms     |  None | 8468.531086910072  |  None  |
|   sqft_living    |  None | 24.420720982445214 |  None  |
|  sqft_living_sqrt|  None | 350.0605533860648  |  None  |
|     sqft_lot     |  None |        0.0         |  None  |
|  sqft_lot_sqrt   |  None |        0.0         |  None  |
|      floors      |  None |        0.0         |  None  |
|  floors_square   |  None |        0.0         |  None  |
|    waterfront    |  None |        0.0         |  None  |
|       view       |  None |        0.0         |  None  |
|    condition     |  None |        0.0         |  None  |
|      grade       |  None | 842.0680348976282  |  None  |
|    sqft_above    |  None | 20.024722417091304 |  None  |
|  sqft_basement   |  None |        0.0         |  None  |
|     yr_built     |  None |        0.0         |  None  |
|   yr_renovated   |  None |        0.0         |  None  |
+------------------+-------+--------------------+--------+
[18 rows x 4 columns]
```

Note that a majority of the weights have been set to zero. So by setting an L1 penalty that's large enough, we are performing a subset selection.

QUIZ QUESTION: According to this list of weights, which of the features have been chosen?

5.2 Selecting an L1 penalty

To find a good L1 penalty, we will explore multiple values using a validation set. Let us do three way split into train, validation, and test sets:

Split our sales data into 2 sets: training and test

Further split our training data into two sets: train, validation

Be *very* careful that you use seed = 1 to ensure you get the same answer!

```
(training_and_validation, testing) = sales.random_split(.9,seed=1) # initial train/test split
(training, validation) = training_and_validation.random_split(0.5, seed=1) # split training into train and validate
```

Next, we write a loop that does the following:

For `l1_penalty` in [10^1, 10^1.5, 10^2, 10^2.5, ..., 10^7] (to get this in Python, type `np.logspace(1, 7, num=13)`.)

Fit a regression model with a given `l1_penalty` on TRAIN data.
Specify `l1_penalty=l1_penalty` and `l2_penalty=0.` in the parameter list.
Compute the RSS on VALIDATION data (here you will want to use `.predict()`) for that `l1_penalty`

Report which `l1_penalty` produced the lowest RSS on validation data.

When you call `linear_regression.create()` make sure you set `validation_set = None`.
Note: you can turn off the print out of `linear_regression.create()` with `verbose = False`

```
import numpy as np
np_logspace = np.logspace(1,7,num=13)
num_penality = len(np_logspace)
RSS = np.empty(num_penality)
for i in range(num_penality):
    model =
turicreate.linear_regression.create(training,target='price',features=all_features,l1_penalty=np_logspace[i],l2_pe
nalty=0,validation_set=None,verbose=False)
    prediction = model.predict(validation)
    errors = prediction - validation['price']
    RSS[i] = np.dot(errors,errors)
print (RSS)
```

```
[6.25766285e+14 6.25766285e+14 6.25766286e+14 6.25766288e+14
 6.25766295e+14 6.25766317e+14 6.25766387e+14 6.25766607e+14
 6.25767303e+14 6.25769508e+14 6.25776518e+14 6.25799063e+14
 6.25883719e+14]
```

```
rss_min = RSS.min()
#print rss_min
for i in range(num_penality):
    #print RSS[i]
    #print np_logspace[i]
    if RSS[i]==rss_min:
        print (np_logspace[i])
```

```
10.0
```

*** QUIZ QUESTION. *** What was the best value for the `l1_penalty`?

```
model_l1 = turicreate.linear_regression.create(training,target='price',features=all_features,l1_penalty=10,l2_penalty=0,validation_set=None,verbose=False)
model_l1.coefficients.print_rows(num_rows=100)
model_l1.coefficients['value'].nnz()
```

```
+------------------+-------+--------------------+--------+
|       name       | index |       value        | stderr |
+------------------+-------+--------------------+--------+
|   (intercept)    |  None |  18993.427212770577  |  None  |
|     bedrooms     |  None |  7936.967679031306   |  None  |
|  bedrooms_square |  None |   936.993368193299   |  None  |
|    bathrooms     |  None |  25409.58893412067   |  None  |
|    sqft_living   |  None |  39.11513637970762   |  None  |
|  sqft_living_sqrt |  None |  1124.6502128077207  |  None  |
|     sqft_lot     |  None | 0.003483618222989743 |  None  |
|   sqft_lot_sqrt  |  None |  148.2583910114082   |  None  |
|      floors      |  None |  21204.33546695013   |  None  |
|   floors_square  |  None |  12915.524336072436  |  None  |
|    waterfront    |  None |  601905.5945452718   |  None  |
|       view       |  None |  93312.85731187189   |  None  |
|    condition     |  None |  6609.035712447216   |  None  |
|      grade       |  None |  6206.939991880551   |  None  |
|    sqft_above    |  None |  43.28705341933558   |  None  |
|  sqft_basement   |  None |  122.36782753411931  |  None  |
|     yr_built     |  None |  9.433635393724884   |  None  |
|   yr_renovated   |  None |  56.072003448822386  |  None  |
+------------------+-------+--------------------+--------+
[18 rows x 4 columns]
```

```
18
```

QUIZ QUESTION Also, using this value of L1 penalty, how many nonzero weights do you have?

```
18
```

Limit the number of nonzero weights

What if we absolutely wanted to limit ourselves to, say, 7 features? This may be important if we want to derive "a rule of thumb" --- an interpretable model that has only a few features in them.
In this section, you are going to implement a simple, two phase procedure to achive this goal:

1. Explore a large range of `l1_penalty` values to find a narrow region of `l1_penalty` values where models are likely to have the desired number of non-zero weights.
2. Further explore the narrow region you found to find a good value for `l1_penalty` that achieves the desired sparsity. Here, we will again use a validation set to choose the best value for `l1_penalty`.

```
max_nonzeros = 7
```

5.3 Exploring the larger range of values to find a narrow range with the desired sparsity

Let's define a wide range of possible `l1_penalty_values`:

```
l1_penalty_values = np.logspace(8, 10, num=20)
```

Now, implement a loop that search through this space of possible `l1_penalty` values:

For `l1_penalty` in `np.logspace(8, 10, num=20)`:

Fit a regression model with a given `l1_penalty` on TRAIN data.

Specify `l1_penalty=l1_penalty` and `l2_penalty=0.` in the parameter list. When you call `linear_regression.create()` make sure you set `validation_set = None`

Extract the weights of the model and count the number of nonzeros. Save the number of nonzeros to a list.

**Hint: `model.coefficients['value']` gives you an SArray with the parameters you learned. If you call the method `.nnz()` on it, you will find the number of non-zero parameters!*

```
np_logspace = np.logspace(8, 10, num=20)
num_penality = len(np_logspace)
RSS = np.empty(num_penality)
for i in range(num_penality):
    model =
turicreate.linear_regression.create(training,target='price',features=all_features,l1_penalty=np_logspace[i],l2_pe
nalty=0,validation_set=None,verbose=False)
    prediction = model.predict(validation)
    errors = prediction - validation['price']
    RSS[i] = np.dot(errors,errors)
    nnz = model.coefficients['value'].nnz()
    print (str(i) + ': '+ str(nnz))
print (RSS)
```

```
# 0: 18
0: 18
1: 18
2: 18
3: 18
4: 17
5: 17
6: 17
7: 17
8: 17
9: 16
10: 15
11: 15
12: 13
13: 12
14: 10
15: 6
16: 5
17: 3
18: 1
19: 1
[6.27492660e+14 6.28210517e+14 6.29176690e+14 6.30650083e+14
```

```
6.32940229e+14 6.36268140e+14 6.41261198e+14 6.48983455e+14
6.60962218e+14 6.77261521e+14 7.01046816e+14 7.37850623e+14
7.96163110e+14 8.69018173e+14 9.66925692e+14 1.08186759e+15
1.24492736e+15 1.38416149e+15 1.23079472e+15 1.22915716e+15]
```

```
print(np_logspace)
```

```
[1.00000000e+08 1.27427499e+08 1.62377674e+08 2.06913808e+08
 2.63665090e+08 3.35981829e+08 4.28133240e+08 5.45559478e+08
 6.95192796e+08 8.85866790e+08 1.12883789e+09 1.43844989e+09
 1.83298071e+09 2.33572147e+09 2.97635144e+09 3.79269019e+09
 4.83293024e+09 6.15848211e+09 7.84759970e+09 1.00000000e+10]
```

Out of this large range, we want to find the two ends of our desired narrow range of `l1_penalty`. At one end, we will have `l1_penalty` values that have too few non-zeros, and at the other end, we will have an `l1_penalty` that has too many non-zeros.

More formally, find:

The largest `l1_penalty` that has more non-zeros than `max_nonzeros` (if we pick a penalty smaller than this value, we will definitely have too many non-zero weights)

Store this value in the variable `l1_penalty_min` (we will use it later)

The smallest `l1_penalty` that has fewer non-zeros than `max_nonzeros` (if we pick a penalty larger than this value, we will definitely have too few non-zero weights)

Store this value in the variable `l1_penalty_max` (we will use it later)

Hint: there are many ways to do this, e.g.:

Programmatically within the loop above

Creating a list with the number of non-zeros for each value of `l1_penalty` and inspecting it to find the appropriate boundaries.

```
l1_penalty_min = np_logspace[14]
l1_penalty_max = np_logspace[15]
print(l1_penalty_min)
print(l1_penalty_max)
```

```
2976351441.6313133
3792690190.7322536
```

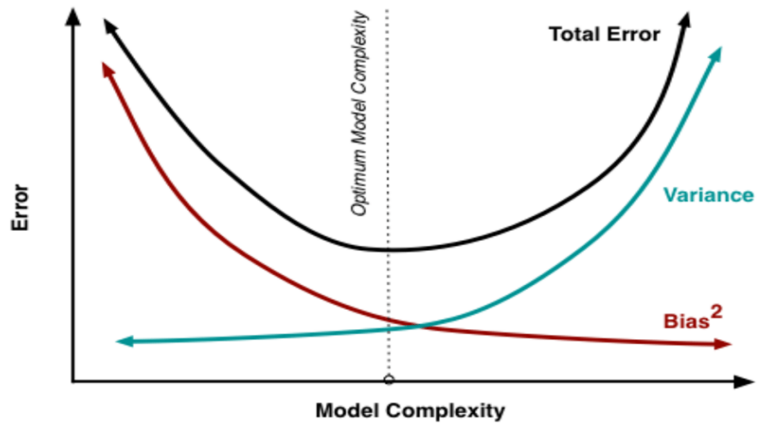


Fig.18

Regression Week 6: Predicting house prices using k-nearest neighbors regression¶

In this notebook, you will implement k-nearest neighbors regression. You will:

Find the k-nearest neighbors of a given query input

Predict the output for the query input using the k-nearest neighbors

Choose the best value of k using a validation set

Fire up Turi Create

```
import turicreate
```

Load in house sales data

For this notebook, we use a subset of the King County housing dataset created by randomly selecting 40% of the houses in the full dataset.

```
sales = turicreate.Sframe('home_data_small.sframe')
```

Import useful functions from previous notebooks

To efficiently compute pairwise distances among data points, we will convert the SFrame into a 2D Numpy array. First import the numpy library and then copy and paste `get_numpy_data()` from the second notebook of Week 2.

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

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 along with the others:
    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_sframe (now including constant):
    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)
```

We will also need the `normalize_features()` function from Week 5 that normalizes all feature columns to unit norm. Paste this function below.

```
def normalize_features(feature_matrix):
```

```
    norms = np.linalg.norm(feature_matrix, axis=0)
    normalized_features = feature_matrix/norms
    return (normalized_features, norms)
```

6.1 Split data into training, test, and validation sets

```
(train_and_validation, test) = sales.random_split(.8, seed=1) # initial train/test split
(train, validation) = train_and_validation.random_split(.8, seed=1) # split training set into training and validation sets
```

Extract features and normalize

Using all of the numerical inputs listed in `feature_list`, transform the training, test, and validation SFrames into Numpy arrays:

```
feature_list = ['bedrooms',
                'bathrooms',
                'sqft_living',
                'sqft_lot',
                'floors',
                'waterfront',
                'view',
                'condition',
                'grade',
                'sqft_above',
                'sqft_basement',
                'yr_built',
                'yr_renovated',
                'lat',
                'long',
                'sqft_living15',
                'sqft_lot15']
features_train, output_train = get_numpy_data(train, feature_list, 'price')
features_test, output_test = get_numpy_data(test, feature_list, 'price')
features_valid, output_valid = get_numpy_data(validation, feature_list, 'price')
```

In computing distances, it is crucial to normalize features. Otherwise, for example, the `sqft_living` feature (typically on the order of thousands) would exert a much larger influence on distance than the `bedrooms` feature (typically on the order of ones). We divide each column of the training feature matrix by its 2-norm, so that the transformed column has unit norm.

IMPORTANT: Make sure to store the norms of the features in the training set. The features in the test and validation sets must be divided by these same norms, so that the training, test, and validation sets are normalized consistently.

```
features_train, norms = normalize_features(features_train) # normalize training set features (columns)
features_test = features_test / norms # normalize test set by training set norms
features_valid = features_valid / norms # normalize validation set by training set norms
```

Compute a single distance

To start, let's just explore computing the "distance" between two given houses. We will take our **query house** to be the first house of the test set and look at the distance between this house and the 10th house of the training set.

To see the features associated with the query house, print the first row (index 0) of the test feature matrix. You should get an 18-dimensional vector whose components are between 0 and 1.

```
print (features_test[0])
```

```
[ 0.01345102  0.01551285  0.01807473  0.01759212  0.00160518  0.017059
  0.        0.05102365  0.0116321   0.01564352  0.01362084  0.02481682
  0.01350306  0.        0.01345386 -0.01346927  0.01375926  0.0016225 ]
```

Now print the 10th row (index 9) of the training feature matrix. Again, you get an 18-dimensional vector with components between 0 and 1.

```
print (features_train[9])
```

```
[ 0.01345102  0.01163464  0.00602491  0.0083488   0.00050756  0.01279425
  0.        0.        0.01938684  0.01390535  0.0096309   0.
  0.01302544  0.        0.01346821 -0.01346254  0.01195898  0.00156612]
```

Compute multiple distances

Of course, to do nearest neighbor regression, we need to compute the distance between our query house and *all* houses in the training set.

To visualize this nearest-neighbor search, let's first compute the distance from our query house (`features_test[0]`) to the first 10 houses of the training set (`features_train[0:10]`) and then search for the nearest neighbor within this small set of houses. Through restricting ourselves to a small set of houses to begin with, we can visually scan the list of 10 distances to verify that our code for finding the nearest neighbor is working.

Write a loop to compute the Euclidean distance from the query house to each of the first 10 houses in the training set.

```
dist_dict = {}
for i in range(0,10):
    dist_dict[i] = np.sqrt(np.sum((features_train[i] - features_test[0])**2))
    print (i, np.sqrt(np.sum((features_train[i] - features_test[0])**2)))
```

```
0 0.0602747091729555
1 0.08546881148827083
2 0.06149946437120284
3 0.05340273978820058
4 0.05844484063938139
5 0.05987921510184001
6 0.05463140497261526
7 0.05543108324159792
8 0.05238362784097273
9 0.05972359371666126
```

*** QUIZ QUESTION ***

Among the first 10 training houses, which house is the closest to the query house?

```
print( min(dist_dict.items(), key=lambda x: x[1]) )
(8, 0.05238362784097273)
```

It is computationally inefficient to loop over computing distances to all houses in our training dataset. Fortunately, many of the Numpy functions can be **vectorized**, applying the same operation over multiple values or vectors. We now walk through this process.

Consider the following loop that computes the element-wise difference between the features of the query house (`features_test[0]`) and the first 3 training houses (`features_train[0:3]`):

```
for i in range(3):
    print (features_train[i]-features_test[0])
    # should print 3 vectors of length 18
[ 0.00000000e+00 -3.87821276e-03 -1.20498190e-02 -1.05552733e-02
  2.08673616e-04 -8.52950206e-03  0.00000000e+00 -5.10236549e-02
  0.00000000e+00 -3.47633726e-03 -5.50336860e-03 -2.48168183e-02
 -1.63756198e-04  0.00000000e+00 -1.70072004e-05  1.30577772e-05
 -5.14364795e-03  6.69281453e-04]
[ 0.00000000e+00 -3.87821276e-03 -4.51868214e-03 -2.26610387e-03
  7.19763456e-04  0.00000000e+00  0.00000000e+00 -5.10236549e-02
  0.00000000e+00 -3.47633726e-03  1.30705004e-03 -1.45830788e-02
 -1.91048898e-04  6.65082271e-02  4.23240653e-05  6.22415897e-06
 -2.89330197e-03  1.47606982e-03]
[ 0.00000000e+00 -7.75642553e-03 -1.20498190e-02 -1.30002801e-02
  1.60518166e-03 -8.52950206e-03  0.00000000e+00 -5.10236549e-02
  0.00000000e+00 -5.21450589e-03 -8.32384500e-03 -2.48168183e-02
 -3.13866046e-04  0.00000000e+00  4.71047219e-05  1.56530415e-05
  3.72914476e-03  1.64764925e-03]
```

The subtraction operator (-) in Numpy is vectorized as follows:

```
print( features_train[0:3] - features_test[0])
[[ 0.00000000e+00 -3.87821276e-03 -1.20498190e-02 -1.05552733e-02
   2.08673616e-04 -8.52950206e-03  0.00000000e+00 -5.10236549e-02
   0.00000000e+00 -3.47633726e-03 -5.50336860e-03 -2.48168183e-02
  -1.63756198e-04  0.00000000e+00 -1.70072004e-05  1.30577772e-05
  -5.14364795e-03  6.69281453e-04]
 [ 0.00000000e+00 -3.87821276e-03 -4.51868214e-03 -2.26610387e-03
   7.19763456e-04  0.00000000e+00  0.00000000e+00 -5.10236549e-02
   0.00000000e+00 -3.47633726e-03  1.30705004e-03 -1.45830788e-02
  -1.91048898e-04  6.65082271e-02  4.23240653e-05  6.22415897e-06
  -2.89330197e-03  1.47606982e-03]
 [ 0.00000000e+00 -7.75642553e-03 -1.20498190e-02 -1.30002801e-02
   1.60518166e-03 -8.52950206e-03  0.00000000e+00 -5.10236549e-02
   0.00000000e+00 -5.21450589e-03 -8.32384500e-03 -2.48168183e-02
  -3.13866046e-04  0.00000000e+00  4.71047219e-05  1.56530415e-05
   3.72914476e-03  1.64764925e-03]]
```

6.2 Perform 1-nearest neighbor regression

Now that we have the element-wise differences, it is not too hard to compute the Euclidean distances between our query house and all of the training houses. First, write a single-line expression to define a variable `diff` such that `diff[i]` gives the element-wise difference between the features of the query house and the `i`-th training house.

```
diff = features_train - features_test[0]
```

To test the code above, run the following cell, which should output a value -0.0934339605842:

```
print (diff[-1].sum()) # sum of the feature differences between the query and last training house
# should print -0.0934339605842
```

```
-0.0934339605841801
```

The next step in computing the Euclidean distances is to take these feature-by-feature differences in `diff`, square each, and take the sum over feature indices. That is, compute the sum of square feature differences for each training house (row in `diff`).

By default, `np.sum` sums up everything in the matrix and returns a single number. To instead sum only over a row or column, we need to specifiy the `axis` parameter described in the `np.sum` documentation. In particular, `axis=1` computes the sum across each row.

Below, we compute this sum of square feature differences for all training houses and verify that the output for the 16th house in the training set is equivalent to having examined only the 16th row of `diff` and computing the sum of squares on that row alone.

```
print (np.sum(diff**2, axis=1)[15]) # take sum of squares across each row, and print the 16th sum
print (np.sum(diff[15]**2)) # print the sum of squares for the 16th row -- should be same as above
```

```
0.003307059028786791
0.0033070590287867904
```

With this result in mind, write a single-line expression to compute the Euclidean distances between the query house and all houses in the training set. Assign the result to a variable `distances`.

Hint: Do not forget to take the square root of the sum of squares.

```
distances = np.sqrt(np.sum(diff**2, axis=1))
```

To test the code above, run the following cell, which should output a value 0.0237082324496:

```
print (distances[100] )# Euclidean distance between the query house and the 101th training house
# should print 0.0237082324496
```

```
0.023708232449603735
```

Now you are ready to write a function that computes the distances from a query house to all training houses. The function should take two parameters: (i) the matrix of training features and (ii) the single feature vector associated with the query.

```
def compute_distances(train_matrix, query_vector):
    diff = train_matrix - query_vector
    distances = np.sqrt(np.sum(diff**2, axis=1))
    return distances
```

*** QUIZ QUESTIONS ***

1. Take the query house to be third house of the test set (`features_test[2]`). What is the index of the house in the training set that is closest to this query house?
2. What is the predicted value of the query house based on 1-nearest neighbor regression?

```
third_house_distance = compute_distances(features_train, features_test[2])
print (third_house_distance.argsort()[:1], min(third_house_distance))
print (third_house_distance[382])
```

```
[382] 0.002860495267507927
0.002860495267507927
```

```
print (np.argsort(third_house_distance, axis = 0)[:4])
print (output_train[382])
```

```
[ 382 1149 4087 3142]
249000
```

6.3 Perform k-nearest neighbor regression

For k-nearest neighbors, we need to find a *set* of k houses in the training set closest to a given query house. We then make predictions based on these k nearest neighbors.

Fetch k-nearest neighbors

Using the functions above, implement a function that takes in

the value of k;

the feature matrix for the training houses; and

the feature vector of the query house

and returns the indices of the k closest training houses. For instance, with 2-nearest neighbor, a return value of [5, 10] would indicate that the 6th and 11th training houses are closest to the query house.

```
def compute_k_nearest_neighbors(k, features_matrix, feature_vector):
    distances = compute_distances(features_matrix, feature_vector)
    return np.argsort(distances, axis = 0)[:k]
```

*** QUIZ QUESTION ***

Take the query house to be third house of the test set (`features_test[2]`). What are the indices of the 4 training houses closest to the query house?

```
print (compute_k_nearest_neighbors(4, features_train, features_test[2]))
```

```
[ 382 1149 4087 3142]
```

Make a single prediction by averaging k nearest neighbor outputs

Now that we know how to find the k-nearest neighbors, write a function that predicts the value of a given query house. **For simplicity, take the average of the prices of the k nearest neighbors in the training set**. The function should have the following parameters:

the value of k;

the feature matrix for the training houses;

the output values (prices) of the training houses; and

the feature vector of the query house, whose price we are predicting.

The function should return a predicted value of the query house.
Hint: You can extract multiple items from a Numpy array using a list of indices. For instance, `output_train[[6, 10]]` returns the prices of the 7th and 11th training houses.

```
def compute_distances_k_avg(k, features_matrix, output_values, feature_vector):
  k_neigbors = compute_k_nearest_neighbors(k, features_matrix, feature_vector)
  avg_value = np.mean(output_values[k_neigbors])
  return avg_value
```

*** QUIZ QUESTION ***

Again taking the query house to be third house of the test set (`features_test[2]`), predict the value of the query house using k-nearest neighbors with `k=4` and the simple averaging method described and implemented above.

```
print (compute_distances_k_avg(4, features_train, output_train, features_test[2]))
413987.5
```

Compare this predicted value using 4-nearest neighbors to the predicted value using 1-nearest neighbor computed earlier.

6.4 Make multiple predictions

Write a function to predict the value of *each and every* house in a query set. (The query set can be any subset of the dataset, be it the test set or validation set.) The idea is to have a loop where we take each house in the query set as the query house and make a prediction for that specific house. The new function should take the following parameters:

the value of k;

the feature matrix for the training houses;

the output values (prices) of the training houses; and

the feature matrix for the query set.

```
print (features_test[0:10].shape[0])
```

10

```
def compute_distances_k_all(k, features_matrix, output_values, feature_vector):
    num_of_rows = feature_vector.shape[0]
    predicted_values = []
    for i in range(num_of_rows):
        avg_value = compute_distances_k_avg(k, features_train, output_train, features_test[i])
        predicted_values.append(avg_value)
    return predicted_values
```

*** QUIZ QUESTION ***

Make predictions for the first 10 houses in the test set using k-nearest neighbors with `k=10`.

1. What is the index of the house in this query set that has the lowest predicted value?
2. What is the predicted value of this house?

```
predicted_values = compute_distances_k_all(10, features_train, output_train, features_test[0:10])
print (predicted_values)
print (predicted_values.index(min(predicted_values)))
```

[881300.0, 431860.0, 460595.0, 430200.0, 766750.0, 667420.0, 350032.0, 512800.7, 484000.0, 457235.0]

= 6

6.5 Choosing the best value of k using a validation set

There remains a question of choosing the value of k to use in making predictions. Here, we use a validation set to choose this value. Write a loop that does the following:

For `k` in [1, 2, ..., 15]:

Makes predictions for each house in the VALIDATION set using the k-nearest neighbors from the TRAINING set.

Computes the RSS for these predictions on the VALIDATION set

Stores the RSS computed above in `rss_all`

Report which `k` produced the lowest RSS on VALIDATION set.

(Depending on your computing environment, this computation may take 10-15 minutes.)

```
rss_all = []
for k in range(1,16):
    predict_value = compute_distances_k_all(k, features_train, output_train, features_valid)
    residual = (output_valid - predict_value)
    rss = sum(residual**2)
    rss_all.append(rss)
```

```
print (rss_all)
```

[355632427476622.0, 317939124951086.5, 313153111376088.5, 301621468995236.0, 294266734341982.4, 287781925015337.9, 287842561046849.3, 286179146468967.94, 281718696883431.6, 280358603702662.75, 278687700531166.9, 278744728841428.25, 275043861135800.9, 273895810640073.47, 272162684453609.75]

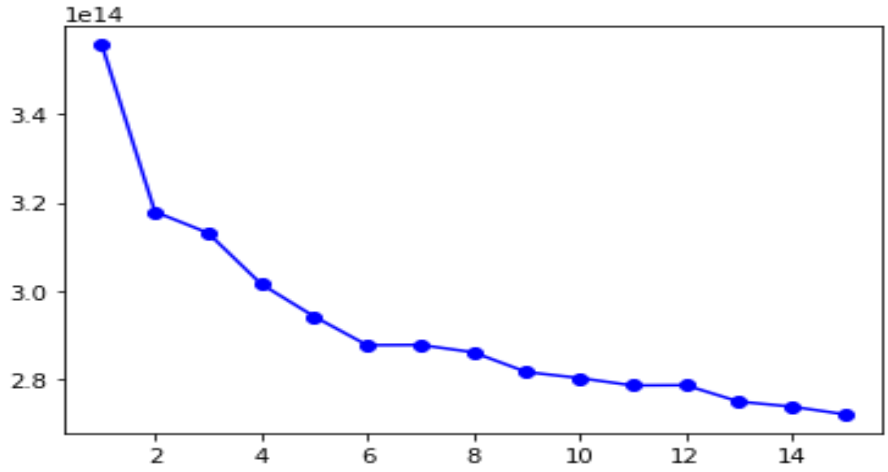
14

To visualize the performance as a function of k, plot the RSS on the VALIDATION set for each considered k value:

```
import matplotlib.pyplot as plt
%matplotlib inline

kvals = range(1, 16)
plt.plot(kvals, rss_all,'bo-')
```

Fig.19



***QUIZ QUESTION ***

What is the RSS on the TEST data using the value of k found above? To be clear, sum over all houses in the TEST set.

```
predict_value = compute_distances_k_all(14, features_train, output_train, features_test)
residual = (output_test - predict_value)
rss = sum(residual**2)
print (rss)
133006256365677.28
```

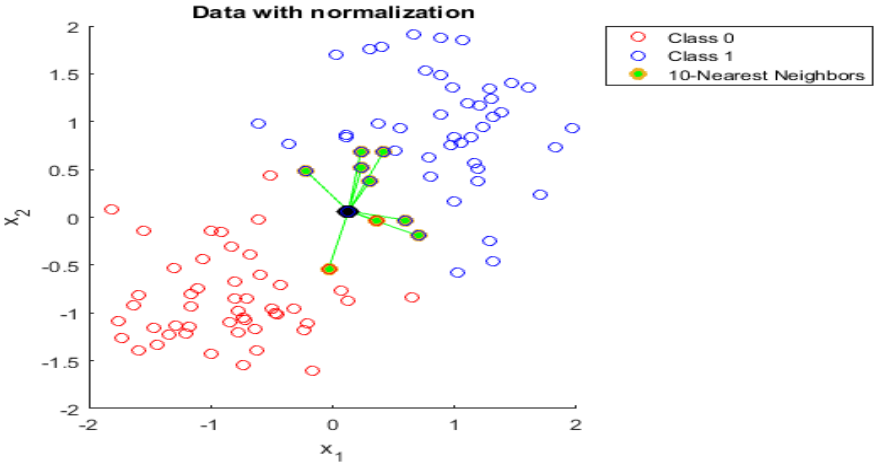


Fig.20

References

The complete training report is based on the online training course from Coursera Machine Learning Course……..https://www.coursera.org

……END……