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 (GraphLab Create) to run polynomial regression
- Use matplotlib to visualize polynomial regressions
- Use a pre-built implementation of regression (GraphLab 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.)

Fire up graphlab create

In [30]:

```
import graphlab
```

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:

In [31]:

```
def polynomial_sframe(feature, degree):
    # assume that degree >= 1
    # initialize the SFrame:
    poly_sframe = graphlab.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
```

Let's use matplotlib to visualize what a polynomial regression looks like on the house data.

In [32]:

```
import matplotlib.pyplot as plt
%matplotlib inline
```
In [33]:
```
sales = graphlab.SFrame('kc_house_data.gl/')
```

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.

In [34]:
```
sales = sales.sort(['sqft_living','price'])
```

Let us revisit the 15th-order polynomial model using the 'sqft_living' input. Generate polynomial features up to degree 15 using `polynomial_sframe()` and fit a model with these features. When fitting the model, use an L2 penalty of `1e-5`:

In [35]:
```
l2_small_penalty = 1e-5
```

Note: When we have so many features and so few data points, the solution can become highly numerically unstable, which can sometimes lead to strange unpredictable results. Thus, rather than using no regularization, we will introduce a tiny amount of regularization (`l2_penalty=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.)

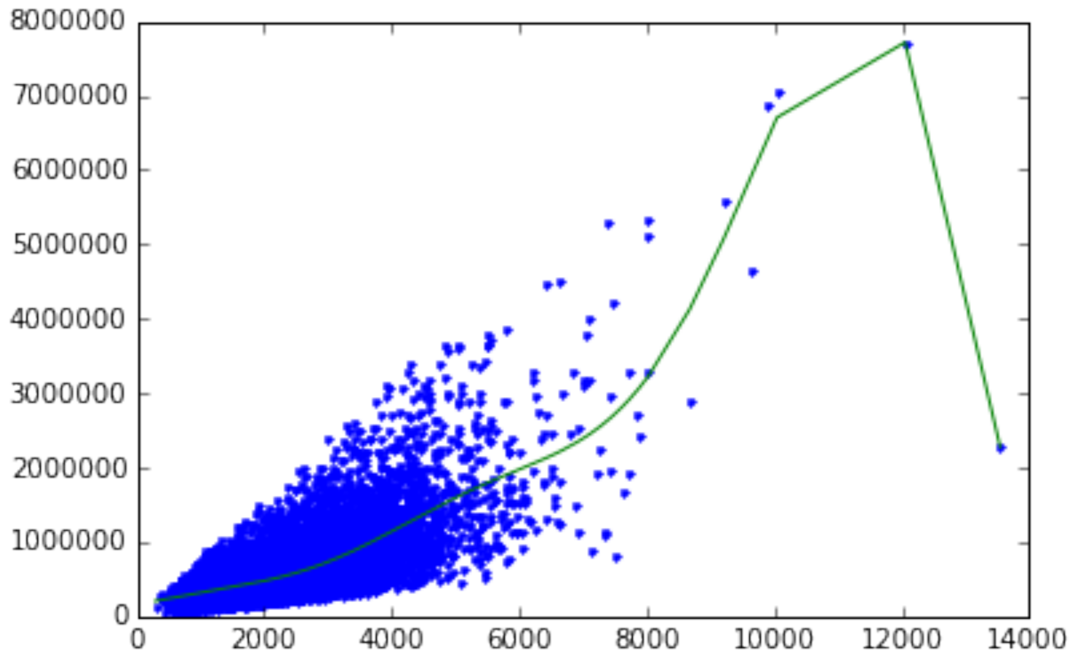
With the L2 penalty specified above, fit the model and print out the learned weights.

Hint: make sure to add 'price' column to the new SFrame before calling `graphlab.linear_regression.create()`. Also, make sure GraphLab Create doesn't create its own validation set by using the option `validation_set=None` in this call.

In [36]:
```
def fit15_deg_poly(data, l2_penalty):
    poly15_data = polynomial_sframe(data['sqft_living'], 15)
    fifteen_features = poly15_data.column_names() # get the name of the
features
    poly15_data['price'] = data['price'] # add price to the data since
it's the target
    model15 = graphlab.linear_regression.create(poly15_data, target =
'price', features = fifteen_features,

l2_penalty=l2_penalty, validation_set = None, verbose=False)

    model15.get("coefficients").print_rows(num_rows=16)

    plt.plot(poly15_data['power_1'],poly15_data['price'],'.',
             poly15_data['power_1'], model15.predict(poly15_data),'-')
```
In [38]:
```
fit15_deg_poly(sales, l2_small_penalty)
```

```
+-------------+-------+--------------------+
|     name    | index |       value        |
+-------------+-------+--------------------+
| (intercept) |  None |   167924.857726    |
|   power_1   |  None |   103.090951289    |
|   power_2   |  None |   0.13460455096    |
|   power_3   |  None | -0.000129071363752 |
```

```
|    power_4    |  None | 5.18928955754e-08   |
|    power_5    |  None | -7.77169299595e-12  |
|    power_6    |  None | 1.71144842837e-16   |
|    power_7    |  None | 4.51177958161e-20   |
|    power_8    |  None | -4.78839816249e-25  |
|    power_9    |  None | -2.33343499941e-28  |
|    power_10   |  None | -7.29022428496e-33  |
|    power_11   |  None | 7.22829146954e-37   |
|    power_12   |  None |  6.9047076722e-41   |
|    power_13   |  None | -3.65843768148e-46  |
|    power_14   |  None | -3.79575941941e-49  |
|    power_15   |  None |  1.1372314991e-53   |
+-------------+-------+--------------------+
[16 rows x 3 columns]
```



QUIZ QUESTION: What's the learned value for the coefficient of feature `power_1`?
Answer: 103.090951289

Observe overfitting¶

Recall from Week 3 that the polynomial fit of degree 15 changed wildly whenever the data changed. In particular, when we split the sales data into four subsets and fit the model of degree 15, the result came out to be very different for each subset. The model had a *high variance*. We will see in a moment that ridge regression reduces such variance. But first, we must reproduce the experiment we did in Week 3.

First, split the data into split the sales data into four subsets of roughly equal size and call them `set_1`, `set_2`, `set_3`, and `set_4`. Use `.random_split` function and make sure you set `seed=0`.

In [39]:

```
(semi_split1, semi_split2) = sales.random_split(.5,seed=0)
```

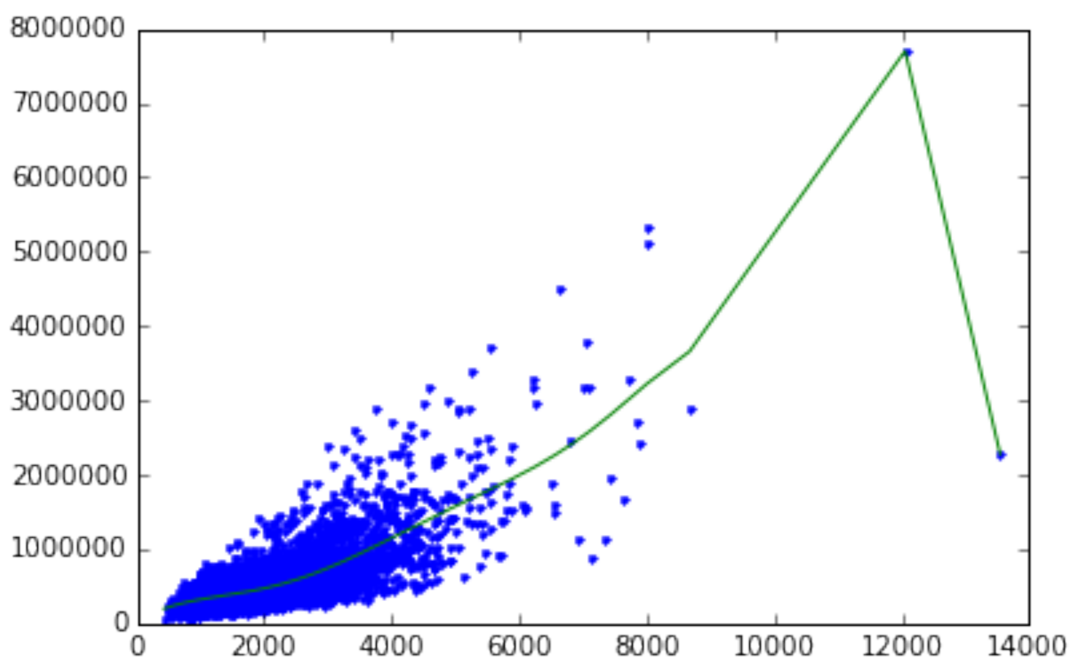
```
(set_1, set_2) = semi_split1.random_split(0.5, seed=0)
(set_3, set_4) = semi_split2.random_split(0.5, seed=0)
```

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 `graphlab.linear_regression.create()`, use the same L2 penalty as before (i.e. `l2_small_penalty`). Also, make sure GraphLab Create doesn't create its own validation set by using the option `validation_set = None` in this call.

In [40]:

```
fit15_deg_poly(set_1, l2_small_penalty)
```

```
+-------------+-------+--------------------+
|     name    | index |       value        |
+-------------+-------+--------------------+
| (intercept) |  None |   9306.46397814    |
|   power_1   |  None |    585.86581347    |
|   power_2   |  None |  -0.397305884724   |
|   power_3   |  None | 0.000141470894825  |
|   power_4   |  None | -1.52945974394e-08 |
|   power_5   |  None | -3.79756526062e-13 |
|   power_6   |  None |  5.9748184732e-17  |
|   power_7   |  None | 1.06888505979e-20  |
|   power_8   |  None | 1.59344052349e-25  |
|   power_9   |  None | -6.9283495515e-29  |
|   power_10  |  None | -6.83813368045e-33 |
|   power_11  |  None | -1.62686203908e-37 |
|   power_12  |  None | 2.85118784287e-41  |
|   power_13  |  None | 3.79998146917e-45  |
|   power_14  |  None | 1.52652617058e-49  |
|   power_15  |  None | -2.33807310252e-53 |
+-------------+-------+--------------------+
[16 rows x 3 columns]
```

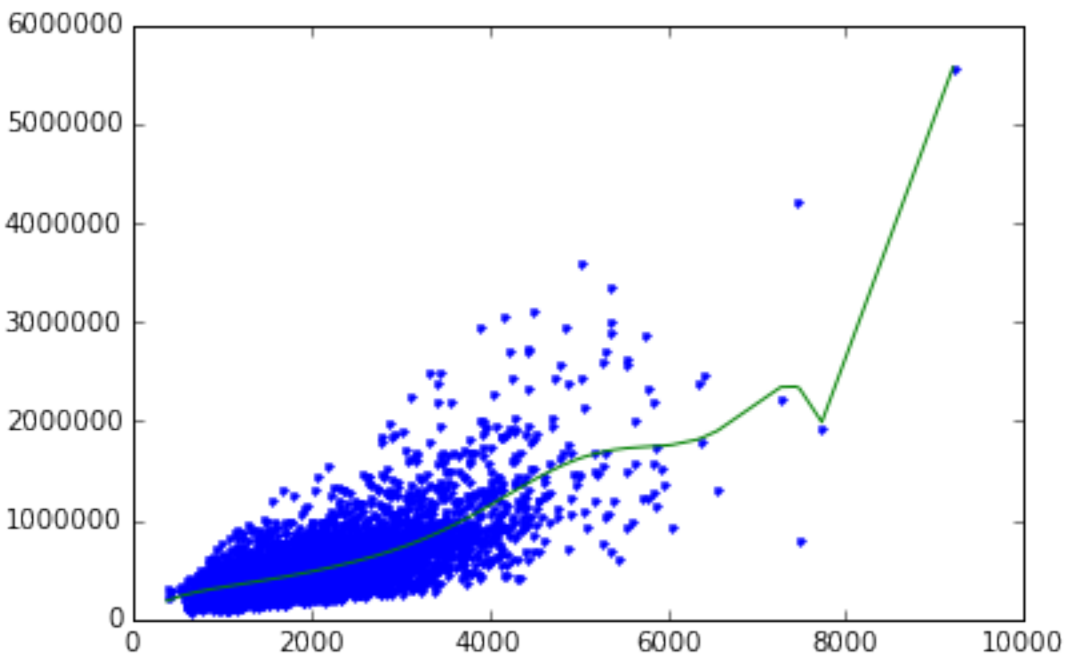


In [41]:

```
fit15_deg_poly(set_2, l2_small_penalty)
```

name	index	value
(intercept)	None	-25115.9059869
power_1	None	783.493802508
power_2	None	-0.767759300173
power_3	None	0.000438766361934
power_4	None	-1.15169161152e-07
power_5	None	6.84281148707e-12
power_6	None	2.5119522464e-15
power_7	None	-2.06440624344e-19
power_8	None	-4.59673058828e-23
power_9	None	-2.71277342492e-29
power_10	None	6.21818505057e-31
power_11	None	6.51741311744e-35
power_12	None	-9.41316275987e-40
power_13	None	-1.02421363129e-42
power_14	None	-1.00391099753e-46
power_15	None	1.30113366848e-50

[16 rows x 3 columns]



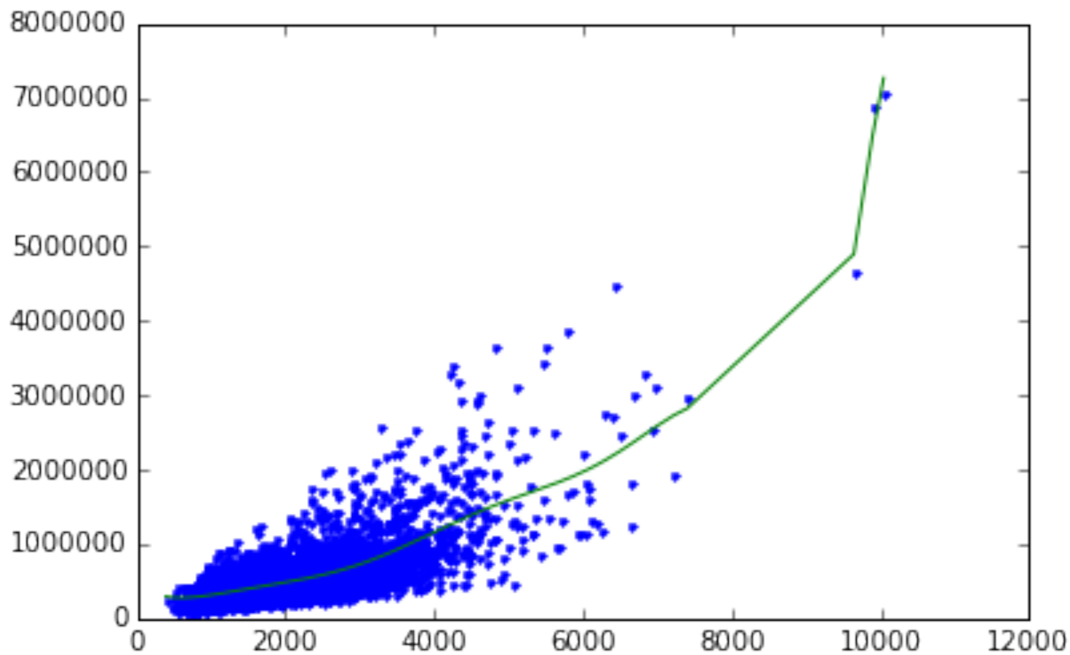
In [42]:

```
fit15_deg_poly(set_3, l2_small_penalty)
```

name	index	value
(intercept)	None	462426.565731
power_1	None	-759.251842854
power_2	None	1.0286700473

power_3	None	-0.000528264527386
power_4	None	1.15422908385e-07
power_5	None	-2.26095948062e-12
power_6	None	-2.08214287571e-15
power_7	None	4.08770475709e-20
power_8	None	2.570791329e-23
power_9	None	1.24311265196e-27
power_10	None	-1.72025834939e-31
power_11	None	-2.96761071315e-35
power_12	None	-1.06574890499e-39
power_13	None	2.42635621458e-43
power_14	None	3.5559876473e-47
power_15	None	-2.85777468723e-51

[16 rows x 3 columns]

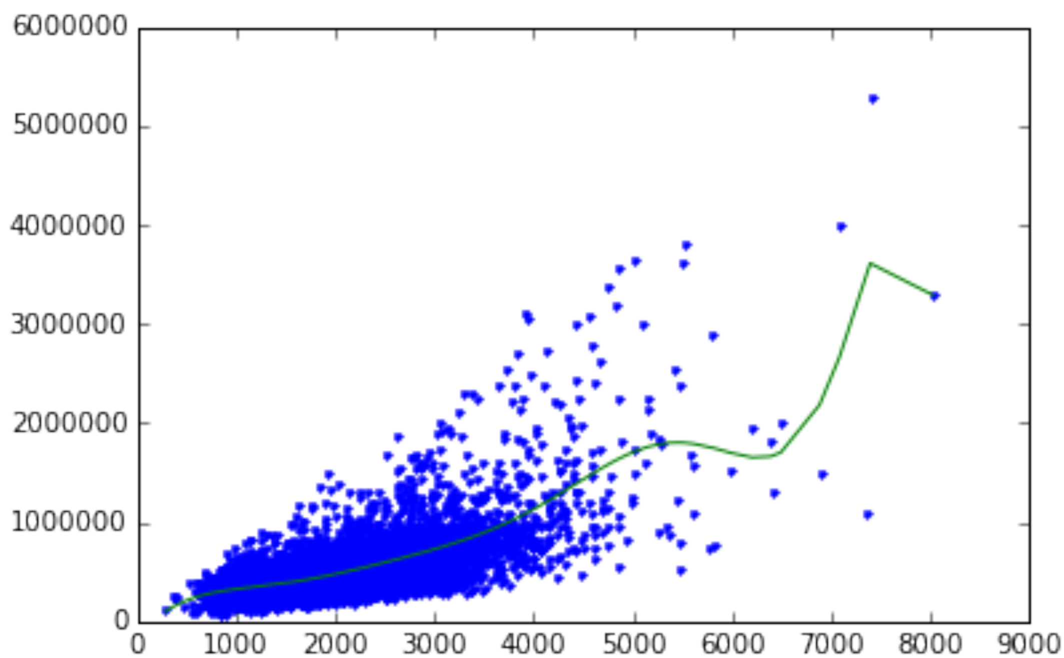


In [43]:

```
fit15_deg_poly(set_4, l2_small_penalty)
```

name	index	value
(intercept)	None	-170240.034791
power_1	None	1247.59035088
power_2	None	-1.2246091264
power_3	None	0.000555254626787
power_4	None	-6.38262361929e-08
power_5	None	-2.20215996475e-11
power_6	None	4.81834697594e-15
power_7	None	4.2146163248e-19
power_8	None	-7.99880749051e-23
power_9	None	-1.32365907706e-26
power_10	None	1.60197797139e-31
power_11	None	2.39904337326e-34

```
|    power_12   |  None | 2.33354505765e-38  |
|    power_13   |  None | -1.79874055895e-42 |
|    power_14   |  None | -6.02862682894e-46 |
|    power_15   |  None | 4.39472672531e-50  |
+-------------+-------+--------------------+
[16 rows x 3 columns]
```



The four curves should differ from one another a lot, as should the coefficients you learned.
QUIZ QUESTION: For the models learned in each of these training sets, what are the smallest and largest values you learned for the coefficient of feature `power_1`? (For the purpose of answering this question, negative numbers are considered "smaller" than positive numbers. So -5 is smaller than -3, and -3 is smaller than 5 and so forth.)
Answer: Min: -759.251842854, Max: 1247.59035088

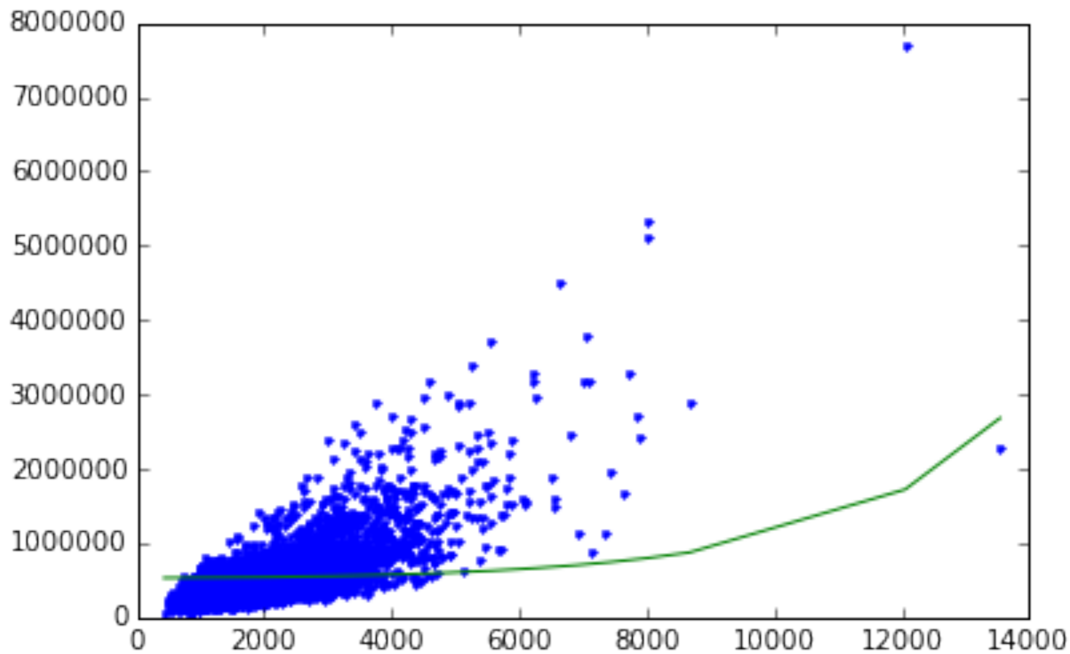
Ridge regression comes to rescue

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 GraphLab Create doesn't create its own validation set by using the option `validation_set = None` in this call.
In [44]:
```
l2_large_penalty = 1e5
```
In [45]:
```
fit15_deg_poly(set_1, l2_large_penalty)
```

```
+-------------+-------+-------------------+
```

name	index	value
(intercept)	None	530317.024516
power_1	None	2.58738875673
power_2	None	0.00127414400592
power_3	None	1.74934226932e-07
power_4	None	1.06022119097e-11
power_5	None	5.42247604482e-16
power_6	None	2.89563828343e-20
power_7	None	1.65000666351e-24
power_8	None	9.86081528409e-29
power_9	None	6.06589348254e-33
power_10	None	3.7891786887e-37
power_11	None	2.38223121312e-41
power_12	None	1.49847969215e-45
power_13	None	9.39161190285e-50
power_14	None	5.84523161981e-54
power_15	None	3.60120207203e-58

[16 rows x 3 columns]

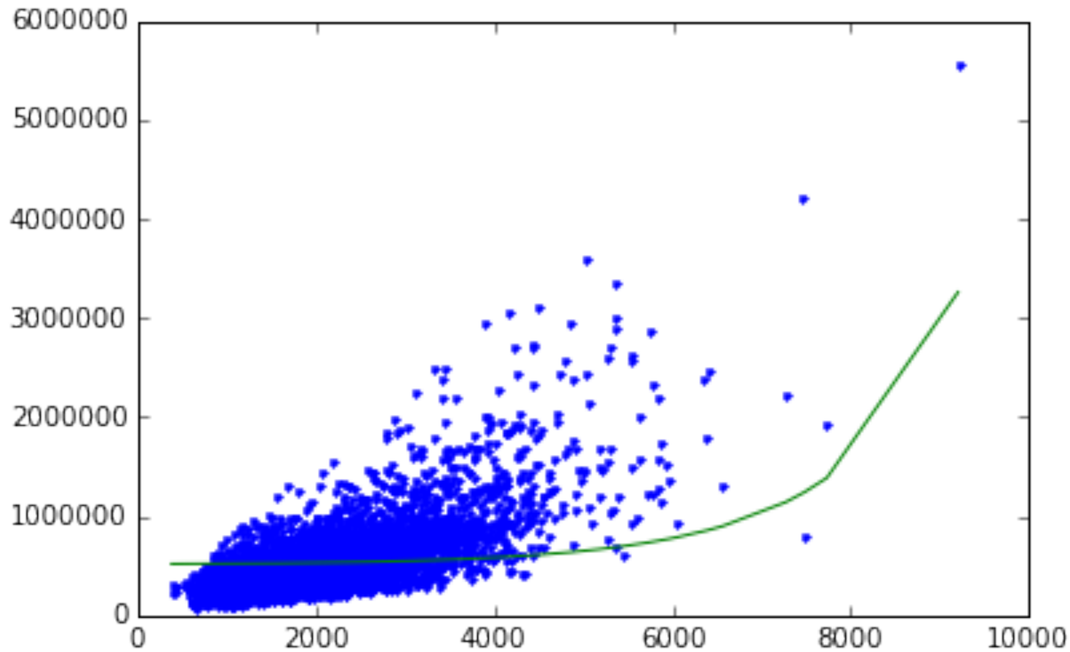


In [46]:
```
fit15_deg_poly(set_2, l2_large_penalty)
```

name	index	value
(intercept)	None	519216.897383
power_1	None	2.04470474182
power_2	None	0.0011314362684
power_3	None	2.93074277549e-07
power_4	None	4.43540598453e-11
power_5	None	4.80849112204e-15
power_6	None	4.53091707826e-19

power_7	None	4.16042910575e-23
power_8	None	3.90094635128e-27
power_9	None	3.7773187602e-31
power_10	None	3.76650326842e-35
power_11	None	3.84228094754e-39
power_12	None	3.98520828414e-43
power_13	None	4.18272762394e-47
power_14	None	4.42738332878e-51
power_15	None	4.71518245412e-55

[16 rows x 3 columns]

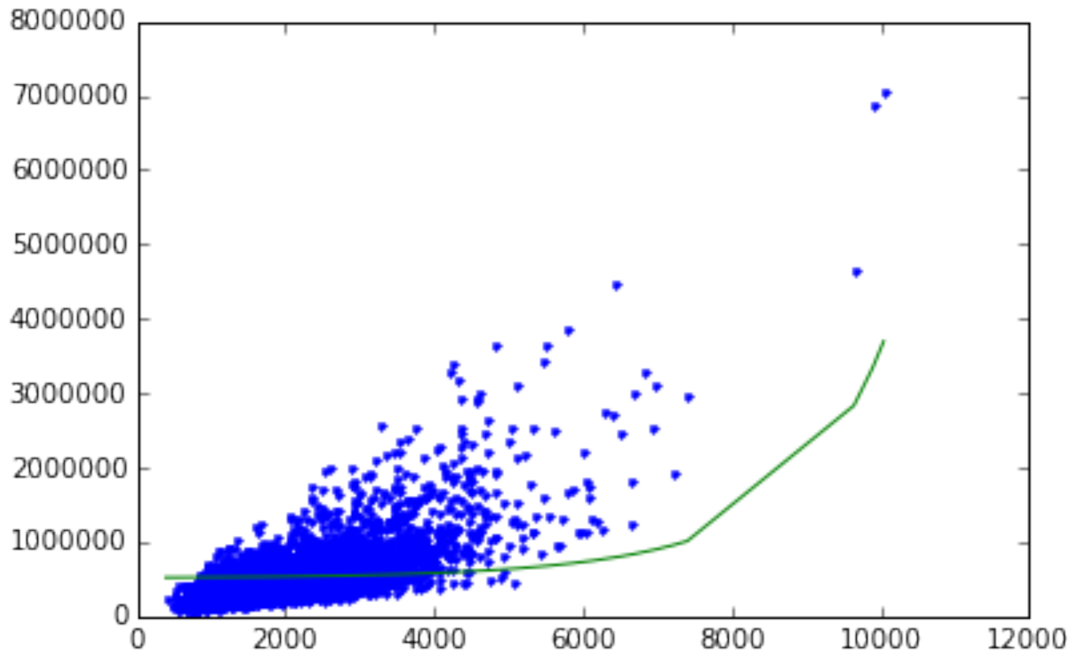


In [47]:

```
fit15_deg_poly(set_3, l2_large_penalty)
```

name	index	value
(intercept)	None	522911.518048
power_1	None	2.26890421877
power_2	None	0.00125905041842
power_3	None	2.77552918155e-07
power_4	None	3.2093309779e-11
power_5	None	2.87573572364e-15
power_6	None	2.50076112671e-19
power_7	None	2.24685265906e-23
power_8	None	2.09349983135e-27
power_9	None	2.00435383296e-31
power_10	None	1.95410800249e-35
power_11	None	1.92734119456e-39
power_12	None	1.91483699013e-43
power_13	None	1.91102277046e-47
power_14	None	1.91246242302e-51
power_15	None	1.91699558035e-55

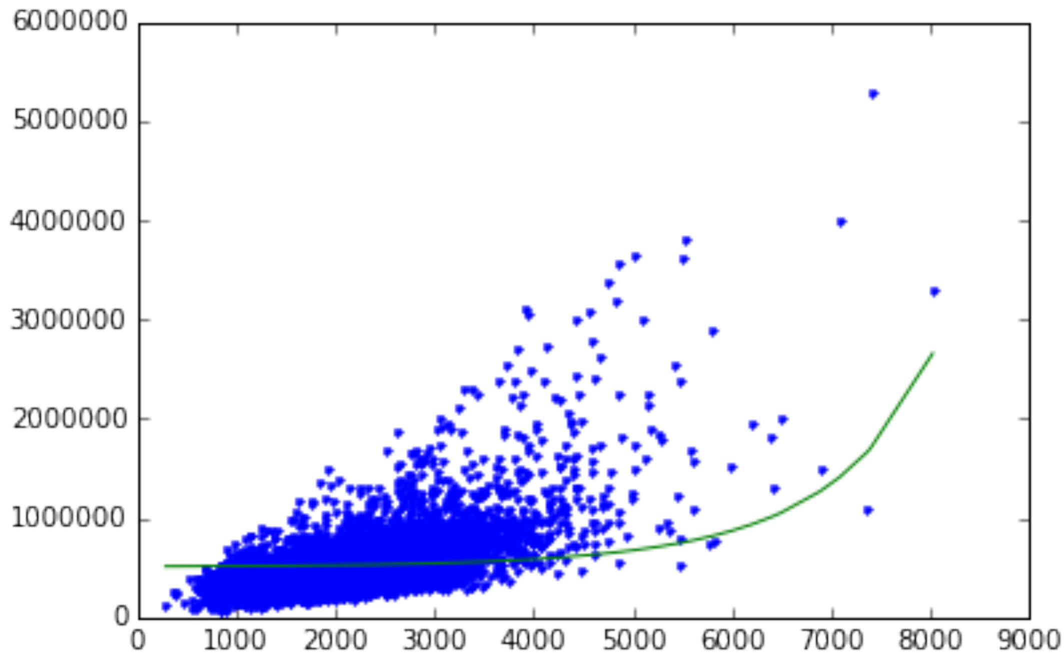
```
+-------------+-------+-------------------+
[16 rows x 3 columns]
```



In [48]:

```
fit15_deg_poly(set_4, l2_large_penalty)
```

```
+-------------+-------+-------------------+
|     name    | index |       value       |
+-------------+-------+-------------------+
| (intercept) |  None |   513667.087087   |
|   power_1   |  None |   1.91040938244   |
|   power_2   |  None |  0.00110058029175 |
|   power_3   |  None | 3.12753987879e-07 |
|   power_4   |  None | 5.50067886825e-11 |
|   power_5   |  None | 7.20467557825e-15 |
|   power_6   |  None | 8.24977249384e-19 |
|   power_7   |  None | 9.06503223498e-23 |
|   power_8   |  None | 9.95683160453e-27 |
|   power_9   |  None | 1.10838127982e-30 |
|   power_10  |  None | 1.25315224143e-34 |
|   power_11  |  None | 1.43600781402e-38 |
|   power_12  |  None |  1.662699678e-42  |
|   power_13  |  None |  1.9398172453e-46 |
|   power_14  |  None |  2.2754148577e-50 |
|   power_15  |  None | 2.67948784897e-54 |
+-------------+-------+-------------------+
[16 rows x 3 columns]
```



These curves should vary a lot less, now that you applied a high degree of regularization.
QUIZ QUESTION: For the models learned with the high level of regularization in each of these training sets, what are the smallest and largest values you learned for the coefficient of feature `power_1`? (For the purpose of answering this question, negative numbers are considered "smaller" than positive numbers. So -5 is smaller than -3, and -3 is smaller than 5 and so forth.)
Answer: Min: 1.91040938244, Max: 2.58738875673

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 roughly equal size. Similar to the validation set method, we measure the validation error with one of the segments designated as the validation set. The major difference is that we repeat the process k times as follows:
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. GraphLab Create 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.)

In [49]:
```
(train_valid, test) = sales.random_split(.9, seed=1)
train_valid_shuffled =
graphlab.toolkits.cross_validation.shuffle(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.

In [50]:
```
n = len(train_valid_shuffled)
k = 10 # 10-fold cross-validation

for i in xrange(k):
    start = (n*i)/k
    end = (n*(i+1))/k-1
    print i, (start, end)
```

```
0 (0, 1938)
1 (1939, 3878)
2 (3879, 5817)
3 (5818, 7757)
4 (7758, 9697)
5 (9698, 11636)
6 (11637, 13576)
7 (13577, 15515)
8 (15516, 17455)
9 (17456, 19395)
```

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.

In [51]:
```
train_valid_shuffled[0:10] # rows 0 to 9
```

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
2780400035	2014-05-05 00:00:00+00:00	665000.0	4.0	2.5	2800.0	5900	1	0
1703050500	2015-03-21 00:00:00+00:00	645000.0	3.0	2.5	2490.0	5978	2	0
5700002325	2014-06-05 00:00:00+00:00	640000.0	3.0	1.75	2340.0	4206	1	0
0475000510	2014-11-18 00:00:00+00:00	594000.0	3.0	1.0	1320.0	5000	1	0
0844001052	2015-01-28 00:00:00+00:00	365000.0	4.0	2.5	1904.0	8200	2	0
2781280290	2015-04-27 00:00:00+00:00	305000.0	3.0	2.5	1610.0	3516	2	0
2214800630	2014-11-05 00:00:00+00:00	239950.0	3.0	2.25	1560.0	8280	2	0
2114700540	2014-10-21 00:00:00+00:00	366000.0	3.0	2.5	1320.0	4320	1	0
2596400050	2014-07-30 00:00:00+00:00	375000.0	3.0	1.0	1960.0	7955	1	0

```
4140900050   2015-01-26 00:00:00+00:00  440000.0       4.0      1.75     2180.0 10200  1        0
view     condition        grade  sqft_above        sqft_basement         yr_builtyr_renovated  zipcode         lat
0        3         8         1660   1140    1963   0        98115  47.68093246
0        3         9         2490   0       2003   0        98074  47.62984888
0        5         7         1170   1170    1917   0        98144  47.57587004
0        4         7         1090   230     1920   0        98107  47.66737217
0        5         7         1904   0       1999   0        98010  47.31068733
0        3         8         1610   0       2006   0        98055  47.44911017
0        4         7         1560   0       1979   0        98001  47.33933392
0        3         6         660    660     1918   0        98106  47.53271982
0        4         7         1260   700     1963   0        98177  47.76407345
2        3         8         2000   180     1966   0        98028  47.76382378
long     sqft_living15    sqft_lot15
-122.28583258          2580.0 5900.0
-122.02177564          2710.0 6629.0
-122.28796     1360.0 4725.0
-122.36472902          1700.0 5000.0
-122.0012452 1560.0 12426.0
-122.1878086 1610.0 3056.0
-122.25864364          1920.0 8120.0
-122.34716948          1190.0 4200.0
-122.36361517          1850.0 8219.0
-122.27022456          2590.0 10445.0
[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`.

In [53]:

```
validation4 = train_valid_shuffled[5818:7758] # rows 5818 to 7757
```

To verify that we have the right elements extracted, run the following cell, which computes the average price of the fourth segment. When rounded to nearest whole number, the average should be $536,234.

In [54]:

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

```
536234
```

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.

In [55]:

```
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)
```

```
+------------+---------------------------+-----------+----------+---------
--+
```

```
|     id     |            date           |   price    | bedrooms |
bathrooms |
+------------+---------------------------+-----------+----------+---------
--+
| 2780400035 | 2014-05-05 00:00:00+00:00 |  665000.0 |   4.0    |    2.5
|
| 1703050500 | 2015-03-21 00:00:00+00:00 |  645000.0 |   3.0    |    2.5
|
| 4139480190 | 2014-09-16 00:00:00+00:00 | 1153000.0 |   3.0    |    3.25
|
| 7237300290 | 2015-03-26 00:00:00+00:00 |  338000.0 |   5.0    |    2.5
|
+------------+---------------------------+-----------+----------+---------
--+
+-------------+----------+--------+------------+------+-----------+-------
+------------+
| sqft_living | sqft_lot | floors | waterfront | view | condition | grade
| sqft_above |
+-------------+----------+--------+------------+------+-----------+-------
+------------+
|    2800.0   |   5900   |   1    |     0      |  0   |     3     |   8
|    1660    |
|    2490.0   |   5978   |   2    |     0      |  0   |     3     |   9
|    2490    |
|    3780.0   |  10623   |   1    |     0      |  1   |     3     |   11
|    2650    |
|    2400.0   |   4496   |   2    |     0      |  0   |     3     |   7
|    2400    |
+-------------+----------+--------+------------+------+-----------+-------
+------------+
+---------------+----------+--------------+---------+-------------+
| sqft_basement | yr_built | yr_renovated | zipcode |     lat     |
+---------------+----------+--------------+---------+-------------+
|      1140     |   1963   |      0       |  98115  | 47.68093246 |
|       0       |   2003   |      0       |  98074  | 47.62984888 |
|      1130     |   1999   |      0       |  98006  | 47.55061236 |
|       0       |   2004   |      0       |  98042  | 47.36923712 |
+---------------+----------+--------------+---------+-------------+
+---------------+---------------+-----+
|      long     | sqft_living15 | ... |
+---------------+---------------+-----+
| -122.28583258 |     2580.0    | ... |
| -122.02177564 |     2710.0    | ... |
| -122.10144844 |     3850.0    | ... |
| -122.12606473 |     1880.0    | ... |
+---------------+---------------+-----+
[4 rows x 21 columns]
```

Extract the remainder of the data after *excluding* fourth segment (segment 3) and assign the subset to `train4`.

In [59]:
```
n = len(train_valid_shuffled)
first = train_valid_shuffled[0:5818]
```

```
last = train_valid_shuffled[7758:n]
train4 = first.append(last)
```

To verify that we have the right elements extracted, run the following cell, which computes the average price of the data with fourth segment excluded. When rounded to nearest whole number, the average should be $539,450.

In [60]:
```
print int(round(train4['price'].mean(), 0))
```

```
539450
```

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

In [90]:
```
import numpy as np
def k_fold_cross_validation(k, l2_penalty, data, output, feature_list):
    n = len(data)
    errors = []
    for i in range(0, k):
        start = (n*i)/k
        end = (n*(i+1))/k-1

        validation_data = poly15_data[start:end+1]

        first = poly15_data[0:start]
        last = poly15_data[end+1:n]
        train_data = first.append(last)

        model = graphlab.linear_regression.create(train_data, target =
output, features = feature_list,

l2_penalty=l2_penalty, validation_set = None, verbose=False)

        # First get the predictions
        predictions = model.predict(validation_data)
        # then compute the residuals (since we are squaring it doesn't
matter which order you subtract)
        residuals = validation_data[output] - predictions
        # square the residuals and add them up
        residuals_squared = residuals * residuals
        RSS = residuals_squared.sum()
        errors.append(RSS)
    average_error = np.mean(errors)
    print("l2_penalty: %s, Average RSS: $%.6f" % (l2_penalty,
average_error))
```

```
    return average_error
```

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!

In [92]:

```
poly15_data = polynomial_sframe(train_valid_shuffled['sqft_living'], 15)
fifteen_features = poly15_data.column_names() # get the name of the features
poly15_data['price'] = train_valid_shuffled['price'] # add price to the data since it's the target

results = []

for l2_penalty in np.logspace(1, 7, num=13):
    average_error = k_fold_cross_validation(10, l2_penalty, train_valid_shuffled, 'price', fifteen_features)
    results.append((l2_penalty, average_error))
```

```
l2_penalty: 10.0, Average RSS: $491826427768998.000000
l2_penalty: 31.6227766017, Average RSS: $287504229919123.375000
l2_penalty: 100.0, Average RSS: $160908965822178.187500
l2_penalty: 316.227766017, Average RSS: $122090967326083.593750
l2_penalty: 1000.0, Average RSS: $121192264451214.875000
l2_penalty: 3162.27766017, Average RSS: $123950009289897.625000
l2_penalty: 10000.0, Average RSS: $136837175247519.031250
l2_penalty: 31622.7766017, Average RSS: $171728094842297.406250
l2_penalty: 100000.0, Average RSS: $229361431260422.687500
l2_penalty: 316227.766017, Average RSS: $252940568728599.843750
l2_penalty: 1000000.0, Average RSS: $258682548441132.343750
l2_penalty: 3162277.66017, Average RSS: $262819399742234.156250
l2_penalty: 10000000.0, Average RSS: $264889015377543.812500
```

QUIZ QUESTIONS: What is the best value for the L2 penalty according to 10-fold validation?
Answer: 1000.0

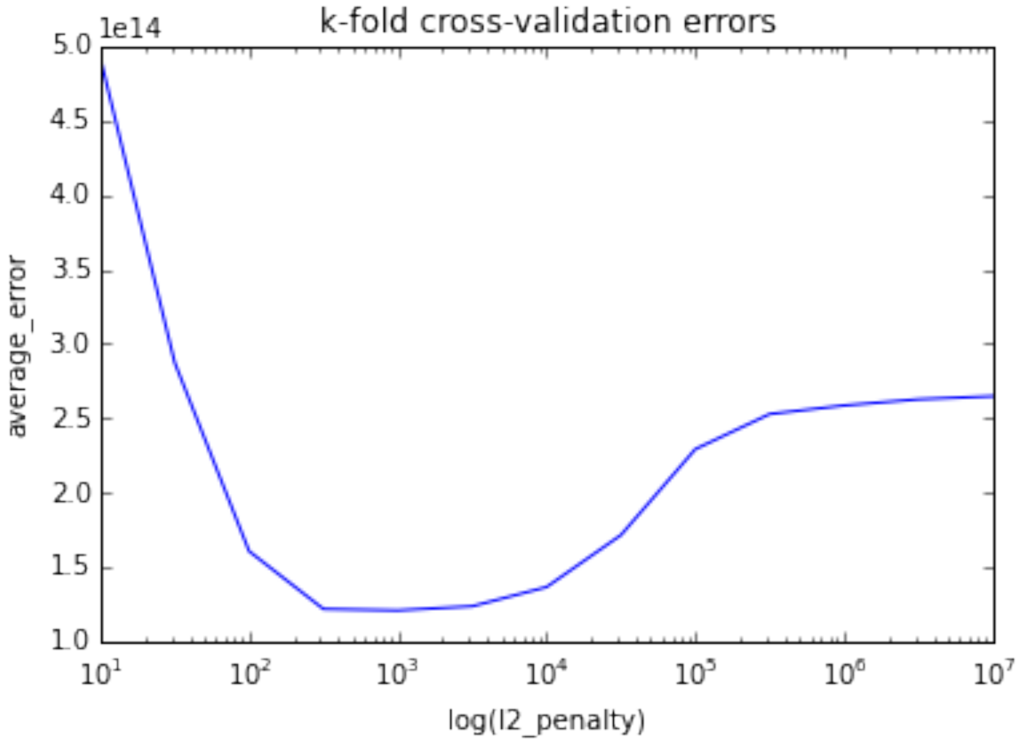
You may find it useful to plot the k-fold cross-validation errors you have obtained to better understand the behavior of the method.

In [93]:

```
plt.plot([x[0] for x in results], [y[1] for y in results],'-')
plt.xscale('log')
plt.xlabel('log(l2_penalty)')
plt.ylabel('average_error')
plt.title('k-fold cross-validation errors')
```

Out[93]:

```
<matplotlib.text.Text at 0x22ef2320>
```



Once you found the best value for the L2 penalty using cross-validation, it is important to retrain a final model on all of the training data using this value of `l2_penalty`. This way, your final model will be trained on the entire dataset.

QUIZ QUESTION: Using the best L2 penalty found above, train a model using all training data. What is the RSS on the TEST data of the model you learn with this L2 penalty?

In [94]:

```
l2_penalty_optimum = 1000.0
model = graphlab.linear_regression.create(poly15_data, target = 'price',
features = fifteen_features,

l2_penalty=l2_penalty_optimum, validation_set = None, verbose=False)
# First get the predictions
predictions = model.predict(test)
# then compute the residuals (since we are squaring it doesn't matter
which order you subtract)
residuals = test['price'] - predictions
# square the residuals and add them up
residuals_squared = residuals * residuals
RSS = residuals_squared.sum()
print("RSS: $%.6f" % (RSS))
```

```
RSS: $252897427447157.500000
```