Overfitting demo

Create a dataset based on a true sinusoidal relationship

Let's look at a synthetic dataset consisting of 30 points drawn from the sinusoid $y = \sin(4x)$:

In [49]:
```
import graphlab
import math
import random
import numpy
from matplotlib import pyplot as plt
%matplotlib inline
```

Create random values for x in interval [0,1)

In [50]:
```
random.seed(98103)
n = 30
x = graphlab.SArray([random.random() for i in range(n)]).sort()
```

Compute y

In [51]:
```
y = x.apply(lambda x: math.sin(4*x))
```

Add random Gaussian noise to y

In [52]:
```
random.seed(1)
e = graphlab.SArray([random.gauss(0,1.0/3.0) for i in range(n)])
y = y + e
```

Put data into an SFrame to manipulate later

In [53]:
```
data = graphlab.SFrame({'X1':x,'Y':y})
data
```

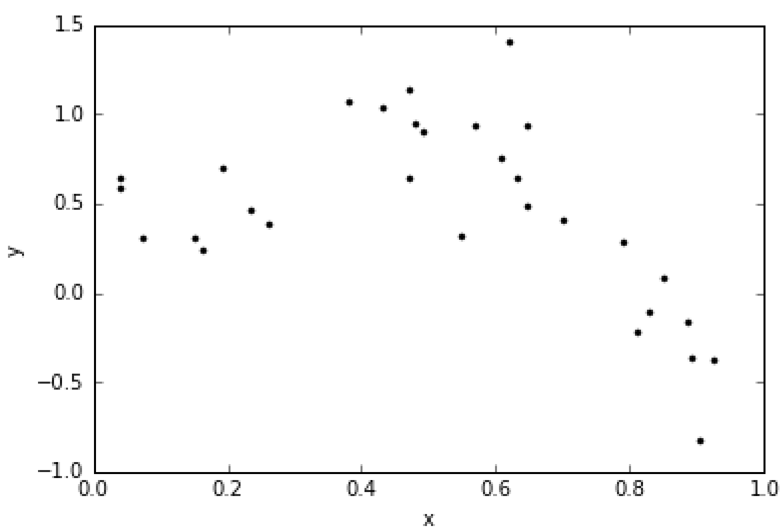
Out[53]:

X1	Y
0.0395789449501	0.587050191026
0.0415680996791	0.648655851372
0.0724319480801	0.307803309485
0.150289044622	0.310748447417
0.161334144502	0.237409625496
0.191956312795	0.705017157224
0.232833917145	0.461716676992
0.259900980166	0.383260507851
0.380145814869	1.06517691429
0.432444723508	1.03184706949

[30 rows x 2 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 a function to plot the data, since we'll do it many times

In [54]:
```
def plot_data(data):
    plt.plot(data['X1'],data['Y'],'k.')
    plt.xlabel('x')
    plt.ylabel('y')

plot_data(data)
```



Define some useful polynomial regression functions

Define a function to create our features for a polynomial regression model of any degree:

In [55]:
```
def polynomial_features(data, deg):
    data_copy=data.copy()
    for i in range(1,deg):
        data_copy['X'+str(i+1)]=data_copy['X'+str(i)]*data_copy['X1']
    return data_copy
```

Define a function to fit a polynomial linear regression model of degree "deg" to the data in "data":

In [56]:
```
def polynomial_regression(data, deg):
    model = graphlab.linear_regression.create(polynomial_features(data,deg),
                                              target='Y', l2_penalty=0.,l1_p
                                              validation_set=None,verbose=Fa
    return model
```

Define function to plot data and predictions made, since we are going to use it many times.

In [57]:
```
def plot_poly_predictions(data, model):
    plot_data(data)

    # Get the degree of the polynomial
    deg = len(model.coefficients['value'])-1

    # Create 200 points in the x axis and compute the predicted value for ea
    x_pred = graphlab.SFrame({'X1':[i/200.0 for i in range(200)]})
    y_pred = model.predict(polynomial_features(x_pred,deg))

    # plot predictions
    plt.plot(x_pred['X1'], y_pred, 'g-', label='degree ' + str(deg) + ' fit'
    plt.legend(loc='upper left')
    plt.axis([0,1,-1.5,2])
```

Create a function that prints the polynomial coefficients in a pretty way :)

In [58]:
```
def print_coefficients(model):
    # Get the degree of the polynomial
    deg = len(model.coefficients['value'])-1

    # Get learned parameters as a list
    w = list(model.coefficients['value'])

    # Numpy has a nifty function to print out polynomials in a pretty way
    # (We'll use it, but it needs the parameters in the reverse order)
    print 'Learned polynomial for degree ' + str(deg) + ':'
    w.reverse()
    print numpy.poly1d(w)
```

Fit a degree-2 polynomial

Fit our degree-2 polynomial to the data generated above:

In [62]:
```
model = polynomial_regression(data, deg=0)
```

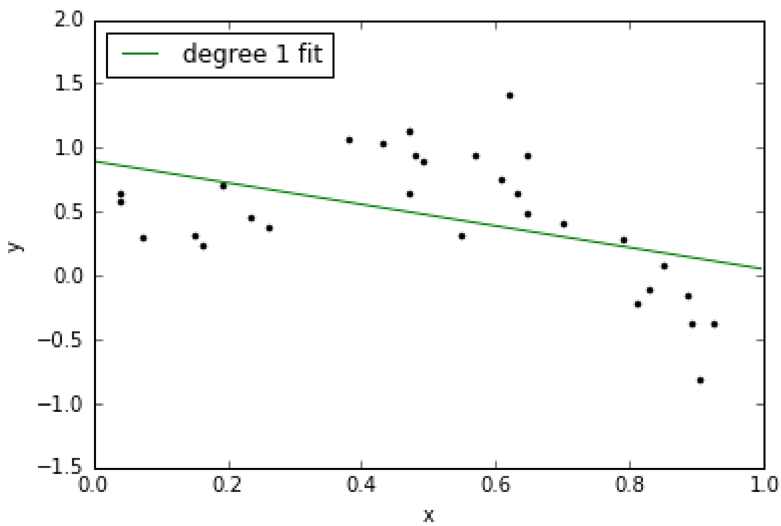
Inspect learned parameters

In [63]:
```
print_coefficients(model)
```

```
Learned polynomial for degree 1:

-0.846 x + 0.8961
```

Form and plot our predictions along a grid of x values:

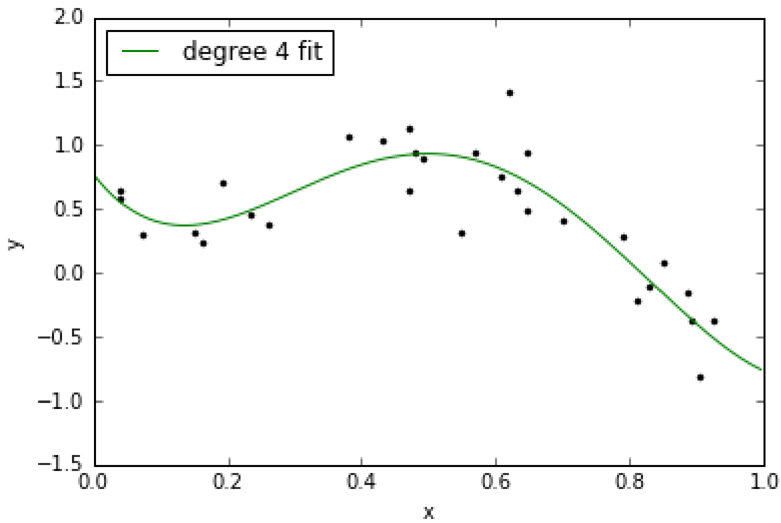
In [64]:
```
plot_poly_predictions(data,model)
```



Fit a degree-4 polynomial

In [43]:
```
model = polynomial_regression(data, deg=4)
print_coefficients(model)
plot_poly_predictions(data,model)
```

```
Learned polynomial for degree 4:
       4         3         2
23.87 x - 53.82 x + 35.23 x - 6.828 x + 0.7755
```



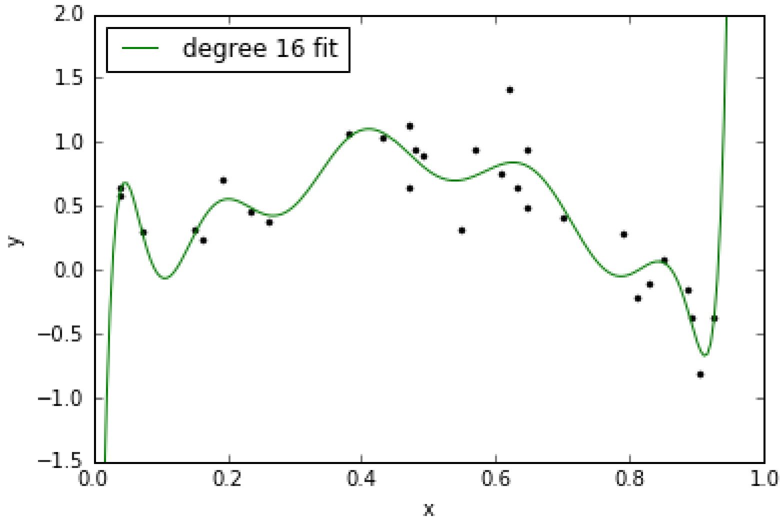
Fit a degree-16 polynomial

In [47]:
```
model = polynomial_regression(data, deg=16)
print_coefficients(model)
```

```
Learned polynomial for degree 16:
           16             15             14             13
2.583e+06 x   - 1.092e+07 x   + 1.443e+07 x   + 1.873e+06 x
              12             11             10             9
 - 2.095e+07 x   + 1.295e+07 x   + 9.366e+06 x   - 1.232e+07 x
              8             7             6             5             4
 - 2.544e+06 x + 1.181e+07 x - 9.325e+06 x + 3.887e+06 x - 9.666e+05 x
              3             2
 + 1.441e+05 x - 1.215e+04 x + 506.6 x - 7.325
```

###Woah!!!! Those coefficients are *crazy*! On the order of 10^6.

In [48]:
```
plot_poly_predictions(data,model)
```



Above: Fit looks pretty wild, too. Here's a clear example of how overfitting is associated with very large magnitude estimated coefficients.

#

#

#

#

Ridge Regression

Ridge regression aims to avoid overfitting by adding a cost to the RSS term of standard least squares that depends on the 2-norm of the coefficients $\|w\|$. The result is penalizing fits with large coefficients. The strength of this penalty, and thus the fit vs. model complexity balance, is controled by a parameter lambda (here called "L2_penalty").

Define our function to solve the ridge objective for a polynomial regression model of any degree:

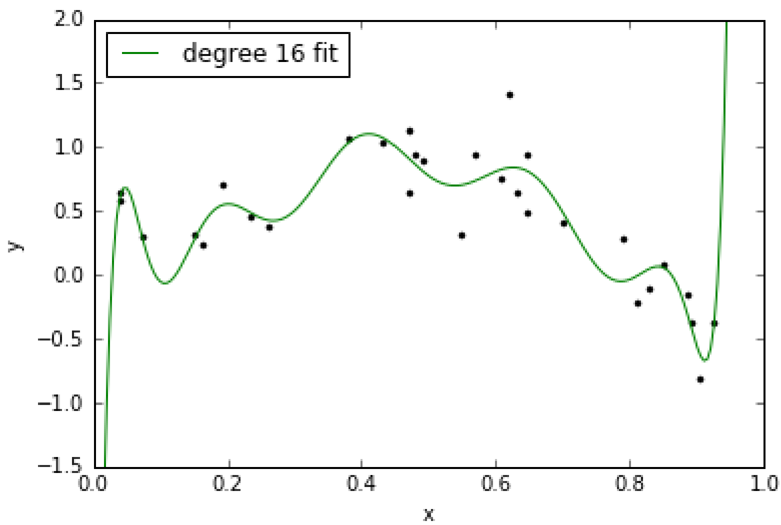
In [24]:
```
def polynomial_ridge_regression(data, deg, l2_penalty):
    model = graphlab.linear_regression.create(polynomial_features(data,deg),
                                              target='Y', l2_penalty=l2_pena
                                              validation_set=None,verbose=Fa
    return model
```

Perform a ridge fit of a degree-16 polynomial using a *very* small penalty strength

In [25]:
```
model = polynomial_ridge_regression(data, deg=16, l2_penalty=1e-25)
print_coefficients(model)
```

```
Learned polynomial for degree 16:
           16             15             14             13
2.583e+06 x  - 1.092e+07 x  + 1.443e+07 x  + 1.873e+06 x
              12             11             10             9
 - 2.095e+07 x  + 1.295e+07 x  + 9.366e+06 x  - 1.232e+07 x
              8             7             6             5             4
 - 2.544e+06 x + 1.181e+07 x - 9.325e+06 x + 3.887e+06 x - 9.666e+05 x
              3             2
 + 1.441e+05 x - 1.215e+04 x + 506.6 x - 7.325
```

In [26]:
```
plot_poly_predictions(data,model)
```

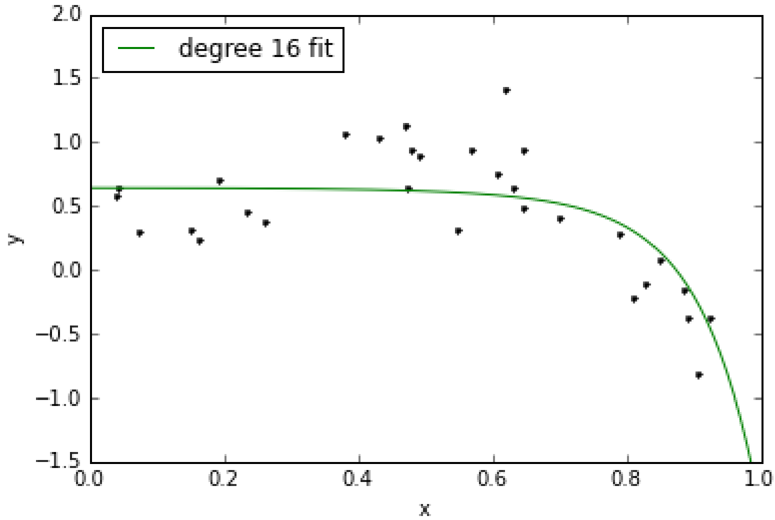


Perform a ridge fit of a degree-16 polynomial using a very large penalty strength

In [20]:
```
model = polynomial_ridge_regression(data, deg=16, l2_penalty=100)
print_coefficients(model)
```

```
Learned polynomial for degree 16:
        16            15            14            13            12           11
-0.301 x  - 0.2802 x  - 0.2604 x  - 0.2413 x  - 0.2229 x  - 0.205 x
           10           9          8          7          6           5
 - 0.1874 x  - 0.1699 x - 0.1524 x - 0.1344 x - 0.1156 x - 0.09534 x
            4            3            2
 - 0.07304 x - 0.04842 x - 0.02284 x - 0.002257 x + 0.6416
```

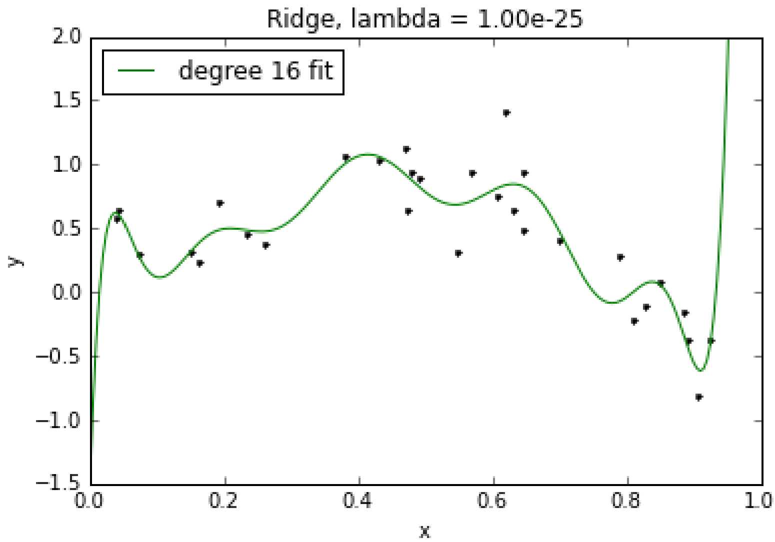
In [21]:
```
plot_poly_predictions(data,model)
```

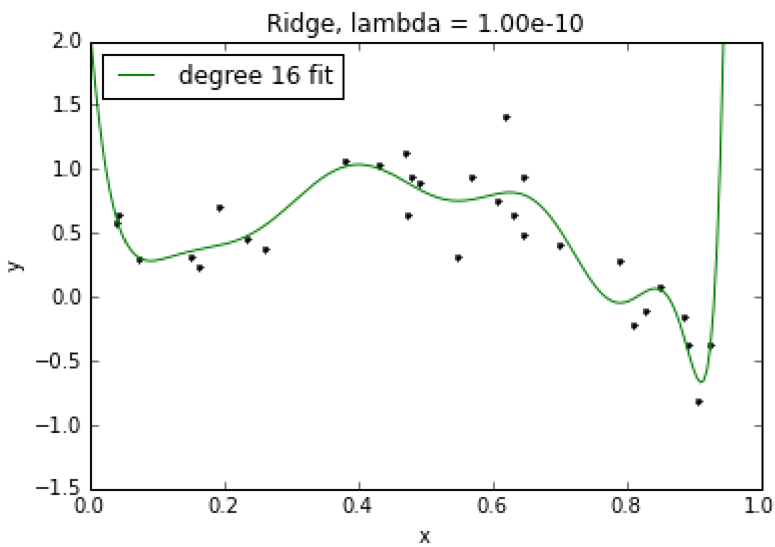


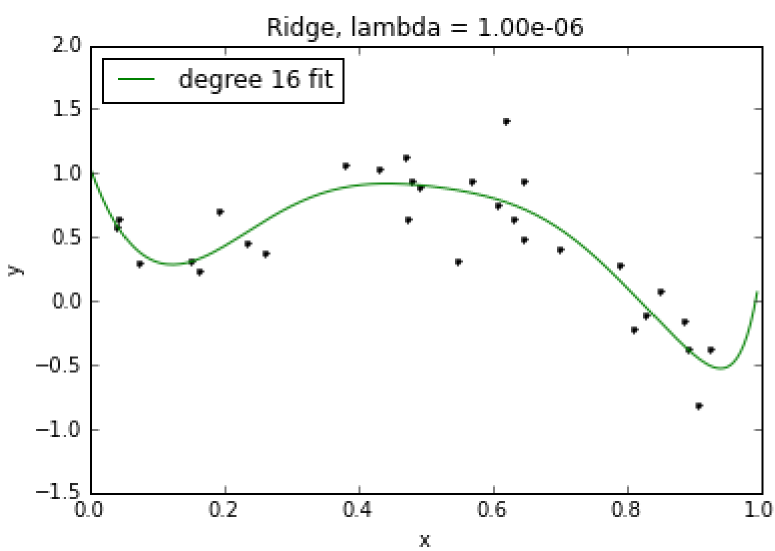
Let's look at fits for a sequence of increasing lambda values

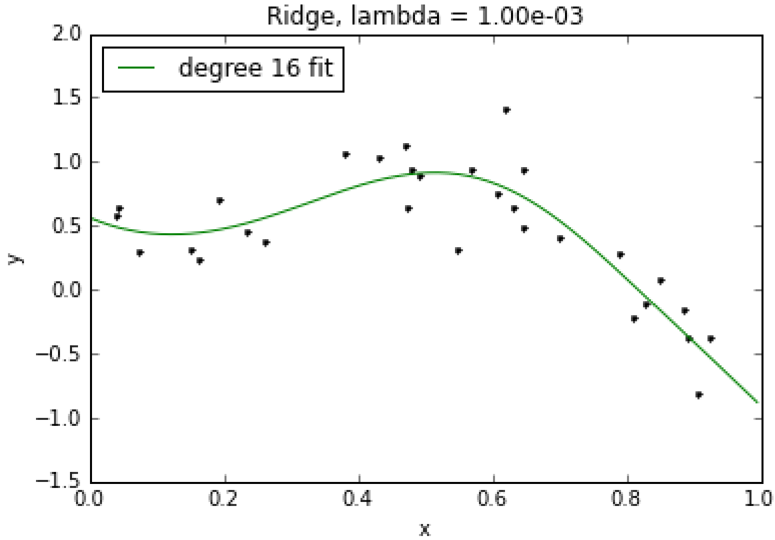
In [22]:
```
for l2_penalty in [1e-25, 1e-10, 1e-6, 1e-3, 1e2]:
    model = polynomial_ridge_regression(data, deg=16, l2_penalty=l2_penalty)
    print 'lambda = %.2e' % l2_penalty
    print_coefficients(model)
    print '\n'
    plt.figure()
    plot_poly_predictions(data,model)
    plt.title('Ridge, lambda = %.2e' % l2_penalty)
```

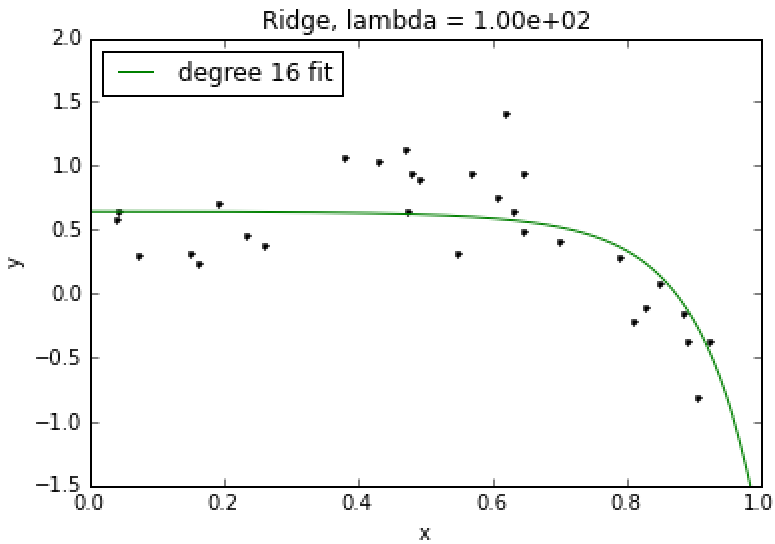
```
lambda = 1.00e-25
Learned polynomial for degree 16:
            16              15              14             13
-4.537e+05 x   + 1.129e+06 x   + 4.821e+05 x   - 3.81e+06 x
              12              11              10              9
 + 3.536e+06 x   + 5.753e+04 x   - 1.796e+06 x   + 2.178e+06 x
              8             7             6             5             4
 - 3.662e+06 x + 4.442e+06 x - 3.13e+06 x + 1.317e+06 x - 3.356e+05 x
             3        2
 + 5.06e+04 x - 4183 x + 160.8 x - 1.621


lambda = 1.00e-10
Learned polynomial for degree 16:
           16              15              14              13
4.975e+04 x   - 7.821e+04 x   - 2.265e+04 x   + 3.949e+04 x
              12         11              10              9              8
 + 4.366e+04 x   + 3074 x   - 3.332e+04 x   - 2.786e+04 x + 1.032e+04 x
              7        6             5             4        3         2
 + 2.962e+04 x - 1440 x - 2.597e+04 x + 1.839e+04 x - 5596 x + 866.1 x -
 65.19 x + 2.159


lambda = 1.00e-06
Learned polynomial for degree 16:
       16         15         14         13         12          11
329.1 x   - 356.4 x   - 264.2 x   + 33.8 x   + 224.7 x   + 210.8 x
          10         9       8         7         6          5          4
 + 49.62 x   - 122.4 x - 178 x - 79.13 x + 84.89 x + 144.9 x + 5.123 x
          3         2
 - 156.9 x + 88.21 x - 14.82 x + 1.059


lambda = 1.00e-03
Learned polynomial for degree 16:
       16         15         14         13         12          11
6.364 x   - 1.596 x   - 4.807 x   - 4.778 x   - 2.776 x   + 0.1238 x
          10         9         8          7          6         5
 + 2.977 x   + 4.926 x + 5.203 x + 3.248 x - 0.9291 x - 6.011 x
          4         3         2
 - 8.395 x - 2.655 x + 9.861 x - 2.225 x + 0.5636


lambda = 1.00e+02
Learned polynomial for degree 16:
        16          15          14          13          12         11
-0.301 x   - 0.2802 x   - 0.2604 x   - 0.2413 x   - 0.2229 x   - 0.205 x
           10          9          8          7          6           5
 - 0.1874 x   - 0.1699 x - 0.1524 x - 0.1344 x - 0.1156 x - 0.09534 x
           4           3           2
 - 0.07304 x - 0.04842 x - 0.02284 x - 0.002257 x + 0.6416
```


Ridge, lambda = 1.00e-25
degree 16 fit
x
y


Ridge, lambda = 1.00e-10
degree 16 fit
x
y


Ridge, lambda = 1.00e-06
degree 16 fit
x
y


Ridge, lambda = 1.00e-03
degree 16 fit
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0
x
y


Ridge, lambda = 1.00e+02
degree 16 fit
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0
x
y

In [23]:
```
data
```

Out[23]:

X1	Y
0.0395789449501	0.587050191026
0.0415680996791	0.648655851372
0.0724319480801	0.307803309485
0.150289044622	0.310748447417
0.161334144502	0.237409625496
0.191956312795	0.705017157224
0.232833917145	0.461716676992
0.259900980166	0.383260507851
0.380145814869	1.06517691429
0.432444723508	1.03184706949

[30 rows x 2 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.

Perform a ridge fit of a degree-16 polynomial using a "good" penalty strength

We will learn about cross validation later in this course as a way to select a good value of the tuning parameter (penalty strength) lambda. Here, we consider "leave one out" (LOO) cross validation, which one can show approximates average mean square error (MSE). As a result, choosing lambda to minimize the LOO error is equivalent to choosing lambda to minimize an approximation to average MSE.

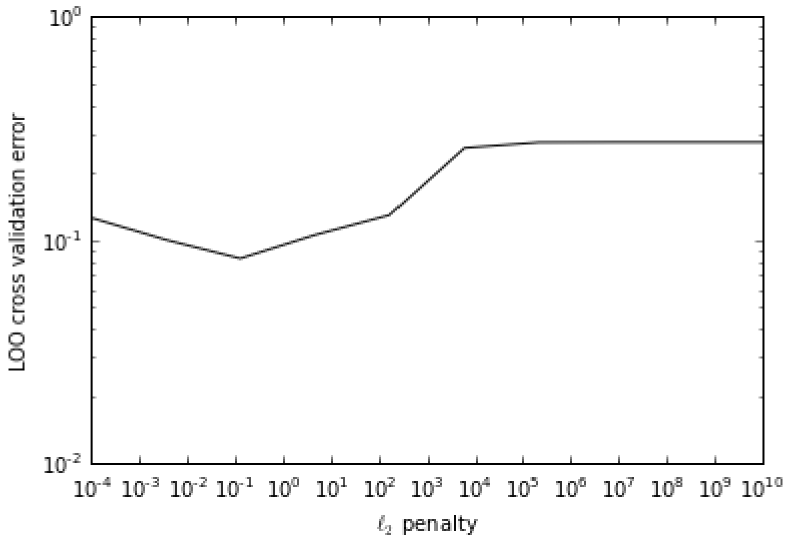
In [30]:
```
# LOO cross validation -- return the average MSE
def loo(data, deg, l2_penalty_values):
    # Create polynomial features
    data = polynomial_features(data, deg)

    # Create as many folds for cross validatation as number of data points
    num_folds = len(data)
    folds = graphlab.cross_validation.KFold(data,num_folds)

    # for each value of l2_penalty, fit a model for each fold and compute av
    l2_penalty_mse = []
    min_mse = None
    best_l2_penalty = None
    for l2_penalty in l2_penalty_values:
        next_mse = 0.0
        for train_set, validation_set in folds:
            # train model
            model = graphlab.linear_regression.create(train_set,target='Y',
                                                      l2_penalty=l2_penalty,
                                                      validation_set=None,ve

            # predict on validation set
            y_test_predicted = model.predict(validation_set)
            # compute squared error
            next_mse += ((y_test_predicted-validation_set['Y'])**2).sum()

        # save squared error in list of MSE for each l2_penalty
        next_mse = next_mse/num_folds
        l2_penalty_mse.append(next_mse)
        if min_mse is None or next_mse < min_mse:
            min_mse = next_mse
            best_l2_penalty = l2_penalty

    return l2_penalty_mse,best_l2_penalty
```

Run LOO cross validation for "num" values of lambda, on a log scale

In [31]:
```
l2_penalty_values = numpy.logspace(-4, 10, num=10)
l2_penalty_mse,best_l2_penalty = loo(data, 16, l2_penalty_values)
```

Plot results of estimating LOO for each value of lambda

In [38]:
```
plt.plot(l2_penalty_values,l2_penalty_mse,'k-')
plt.xlabel('$\ell_2$ penalty')
plt.ylabel('LOO cross validation error')
plt.xscale('log')
plt.yscale('log')
```



Find the value of lambda, λ_{CV}, that minimizes the LOO cross validation error, and plot resulting fit

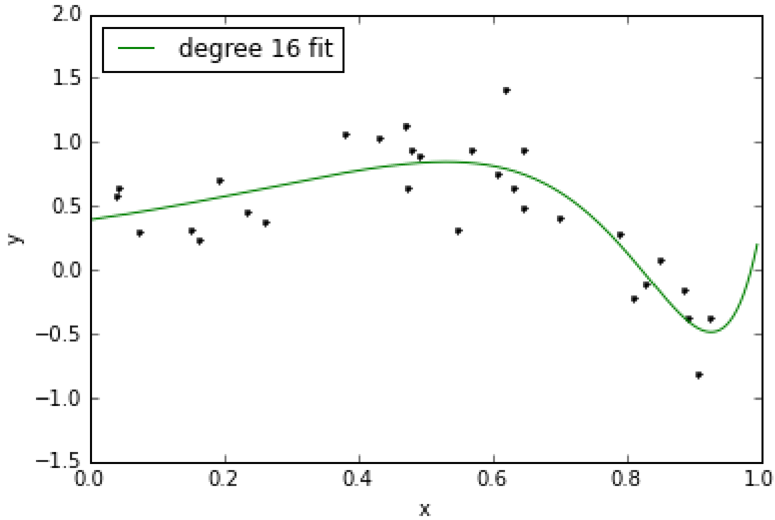
In [39]:
```
best_l2_penalty
```

Out[39]:
```
0.12915496650148839
```

In [34]:
```
model = polynomial_ridge_regression(data, deg=16, l2_penalty=best_l2_penalty
print_coefficients(model)
```

```
Learned polynomial for degree 16:
       16          15           14           13           12            11
1.345 x  + 1.141 x  + 0.9069 x  + 0.6447 x  + 0.3569 x  + 0.04947 x
           10          9          8          7          6          5
 - 0.2683 x  - 0.5821 x - 0.8701 x - 1.099 x - 1.216 x - 1.145 x
          4           3          2
 - 0.7837 x - 0.07406 x + 0.7614 x + 0.7703 x + 0.3918
```

In [35]:
```
plot_poly_predictions(data,model)
```



#

#

#

#

Lasso Regression

Lasso regression jointly shrinks coefficients to avoid overfitting, and implicitly performs feature selection by setting some coefficients exactly to 0 for sufficiently large penalty strength lambda (here called "L1_penalty"). In particular, lasso takes the RSS term of standard least squares and adds a 1-norm cost of the coefficients $\|w\|$.

Define our function to solve the lasso objective for a polynomial regression model of any degree:

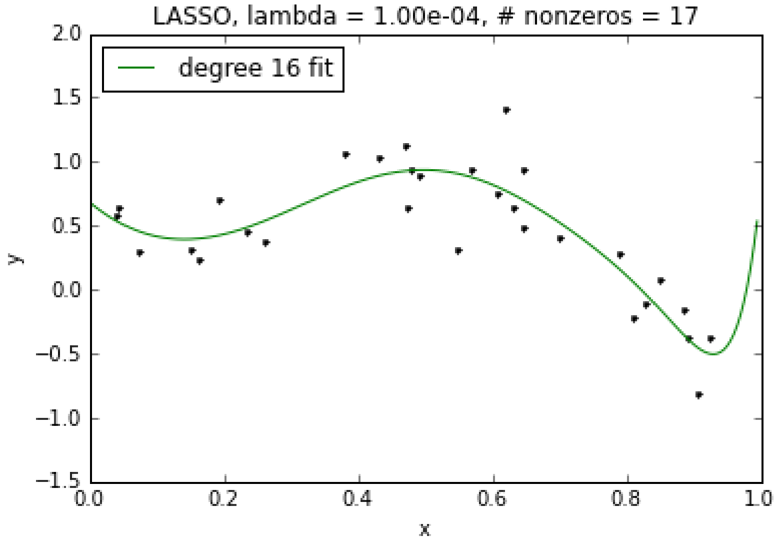
In [36]:
```
def polynomial_lasso_regression(data, deg, l1_penalty):
    model = graphlab.linear_regression.create(polynomial_features(data,deg),
                                              target='Y', l2_penalty=0.,
                                              l1_penalty=l1_penalty,
                                              validation_set=None,
                                              solver='fista', verbose=False,
                                              max_iterations=3000, converger
    return model
```

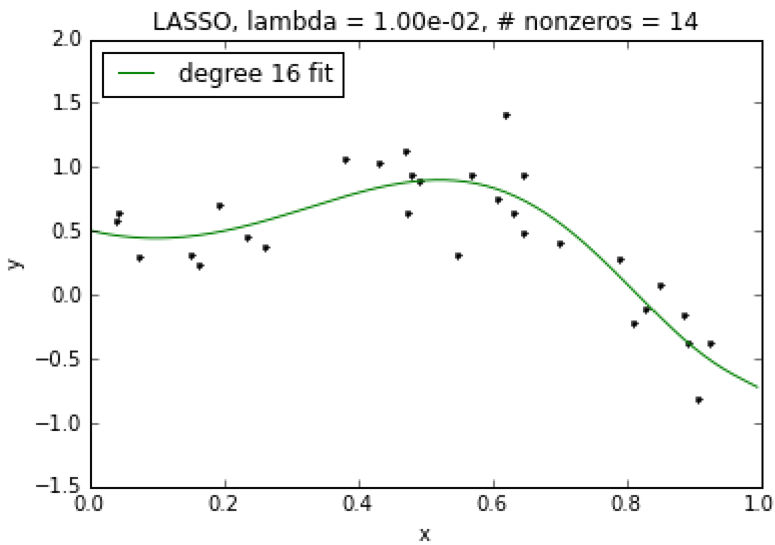
Explore the lasso solution as a function of a few different penalty strengths

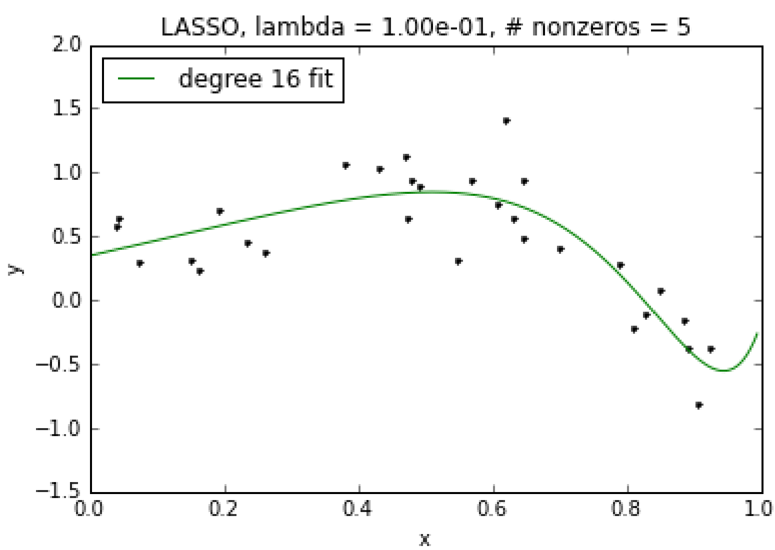
We refer to lambda in the lasso case below as "l1_penalty"

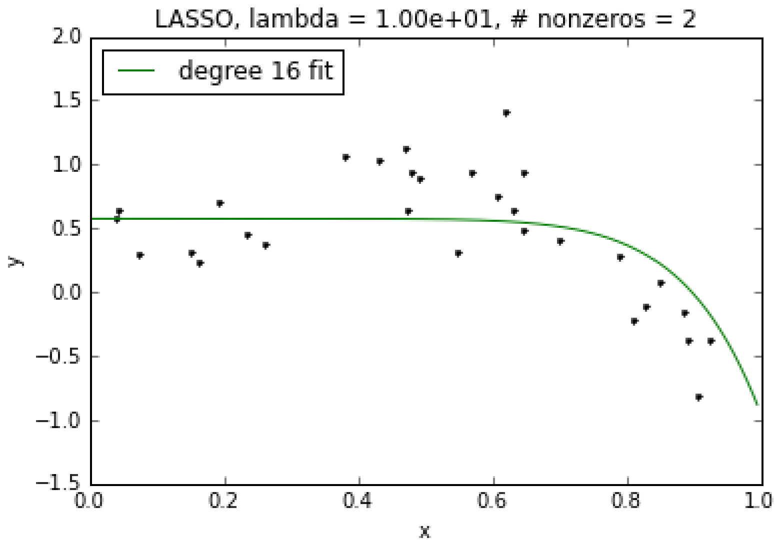
In [37]:
```
for l1_penalty in [0.0001, 0.01, 0.1, 10]:
    model = polynomial_lasso_regression(data, deg=16, l1_penalty=l1_penalty)
    print 'l1_penalty = %e' % l1_penalty
    print 'number of nonzeros = %d' % (model.coefficients['value']).nnz()
    print_coefficients(model)
    print '\n'
    plt.figure()
    plot_poly_predictions(data,model)
    plt.title('LASSO, lambda = %.2e, # nonzeros = %d' % (l1_penalty, (model.
```

```
l1_penalty = 1.000000e-04
number of nonzeros = 17
Learned polynomial for degree 16:
       16        15         14          13          12          11
29.02 x   + 1.35 x   - 12.72 x   - 16.93 x   - 13.82 x   - 6.698 x
          10         9         8          7          6          5
 + 1.407 x   + 8.939 x + 12.88 x + 11.44 x + 3.759 x - 8.062 x
          4         3         2
 - 16.28 x - 7.682 x + 17.86 x - 4.384 x + 0.685


l1_penalty = 1.000000e-02
number of nonzeros = 14
Learned polynomial for degree 16:
       16            15            11            10          9          8
-1.18 x   - 0.001318 x   + 0.08745 x   + 0.7389 x   + 3.828 x + 0.4761 x
           7            6           5         4              3         2
 + 0.1282 x + 0.001952 x - 0.6151 x - 10.11 x - 0.0003954 x + 6.686 x -
 1.28 x + 0.5056


l1_penalty = 1.000000e-01
number of nonzeros = 5
Learned polynomial for degree 16:
      16         6         5
2.21 x   - 1.002 x - 2.962 x + 1.216 x + 0.3473


l1_penalty = 1.000000e+01
number of nonzeros = 2
Learned polynomial for degree 16:
        9
-1.526 x + 0.5755
```


LASSO, lambda = 1.00e-04, # nonzeros = 17
degree 16 fit
y
x


LASSO, lambda = 1.00e-02, # nonzeros = 14
degree 16 fit
y
x


LASSO, lambda = 1.00e-01, # nonzeros = 5
degree 16 fit
y
x



Above: We see that as lambda increases, we get sparser and sparser solutions. However, even for our non-sparse case for lambda=0.0001, the fit of our high-order polynomial is not too wild. This is because, like in ridge, coefficients included in the lasso solution are shrunk relative to those of the least squares (unregularized) solution. This leads to better behavior even without sparsity. Of course, as lambda goes to 0, the amount of this shrinkage decreases and the lasso solution approaches the (wild) least squares solution.

In [ ]:

In [ ]: