When to stop improving: the Bias/Variance tradeoff

In our Linear Regression model we had great sucess when we created second order polynomial features.

We stopped adding features at order 2 because we achieved a perfect in-sample fit.

But if we hadn't: we would not worsen the in-sample fit by continuing to add higher order features.

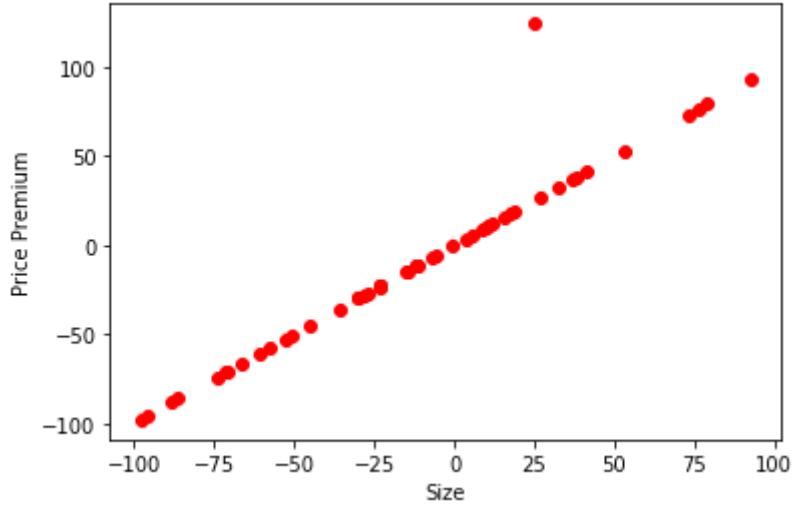
So why not continue to add ?

Let's illustrate the problem: we'll use our first "linear" dataset, but corrupt one point.

This might occur, for example, because of a measurement error.

In [75]:

```
(xlabel, ylabel) = ("Size", "Price Premium")

v1, a1 = 1, .005
lin = recipe_helper.Recipe_Helper(v = v1, a = a1)
X_lin, y_lin = lin.gen_data(num=50)

bad = recipe_helper.Recipe_Helper(v = v1, a = 0)
X_bad, y_bad = bad.gen_data(num=50)

bad_idx = 0
bad.y[bad_idx] = bad.y[bad_idx] * 5
_ = bad.gen_plot(bad.X, bad.y, xlabel, ylabel)
```

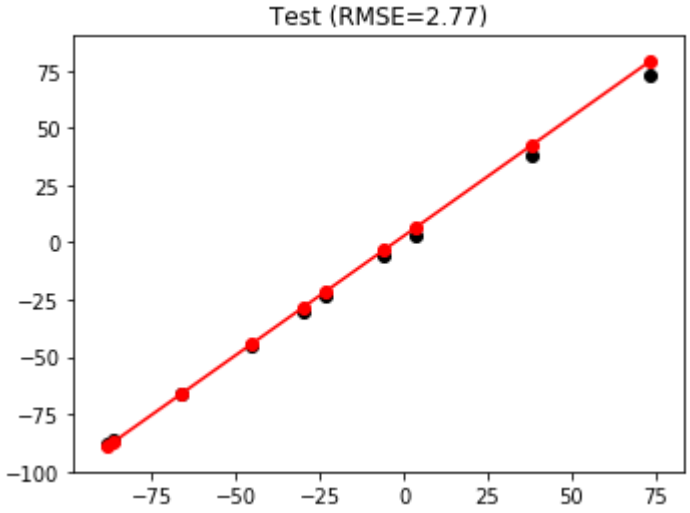


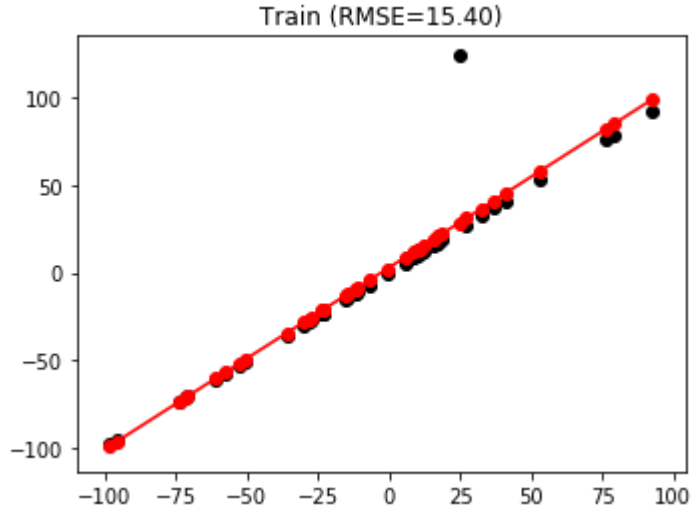
TIP - The time to discover a potential "problem" in the data is during Exploratory Data Analysis - The time to *avoid* the problem is during Prepare the Data: Cleaning - We will show various ways to deal with suspect data - For now: let's deal with the data as is

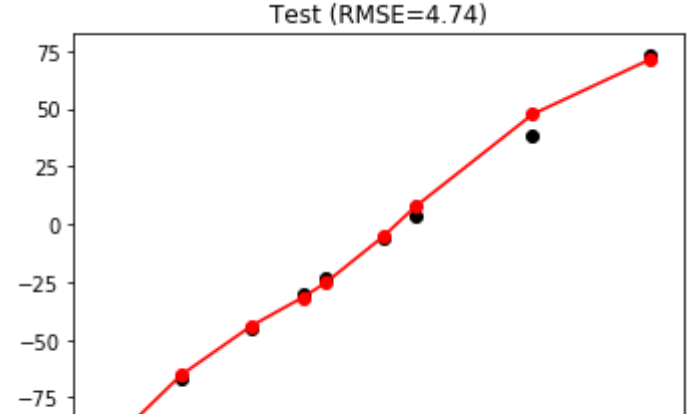
Let's fit a simple linear model (degree 1) and a more complex model (degree 7)

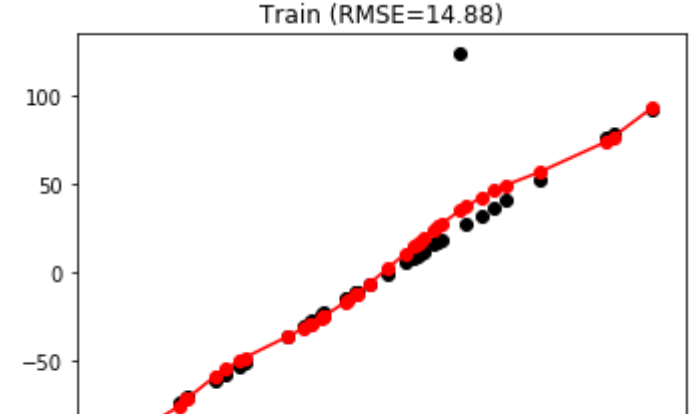
In [74]:

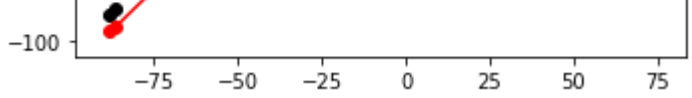
```
from sklearn.pipeline import make_pipeline

for degree in [1, 7]:
    poly_model = make_pipeline(PolynomialFeatures(degree),
                               linear_model.LinearRegression())

    _= bad.run_regress(bad.X, bad.y, model=poly_model, print_summary=False)
```

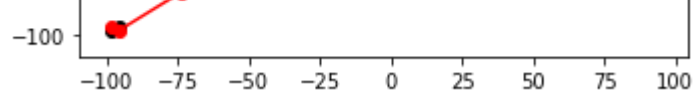









−100
−75
−50
−25
0
25
50
75


−100
−100
−75
−50
−25
0
25
50
75
100

In-sample (training) the Performance Metric decreases with increased model complexity.

But our ultimate goal, the out of sample Performance Metric, is worse.

The more complex model ignores the true linear relationship in a quest to match the training data.

If you add features, you might get a training fit which is better numerically but also misses the essence.

Here is an even more extreme example: fitting a simple to describe function ($\sin(x)$) with a complex polynomial.

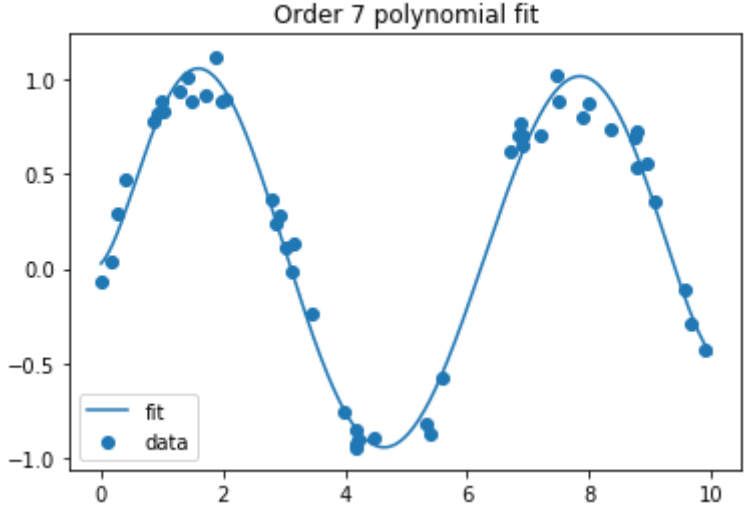
It can be done, but we really are forcing a square peg into a round hole.

In [4]:

```
# Fit a higher order (degree) polynomial
xfit = np.linspace(0, 10, 1000)

from sklearn.preprocessing import PolynomialFeatures

from sklearn.pipeline import make_pipeline
degree = 7
poly_model = make_pipeline(PolynomialFeatures(degree),
                           linear_model.LinearRegression())
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

_= poly_model.fit(x[:, np.newaxis], y)
```

In [5]:

```
# Plot the fit of the high order polynomial
yfit = poly_model.predict(xfit[:, np.newaxis])

_= plt.scatter(x, y, label="data")
_= plt.plot(xfit, yfit, label="fit");
_= plt.title("Order 7 polynomial fit")
_= plt.legend()
```



Even worse: suppose the data's true functional form was simple, but there were a handful of "noisy" mis-measured target values.

Giving ourselves the freedom of adding an arbitrary number of features would cause us to alter the fit in order to match the "noise" rather than the true functional forms.

This would likely lead to worse generalization.

By adding more features, we can improve the in-sample Performance Metric

- eventually we can "memorize" the training set
- but out of sample generaliation would likely suffer
- this is called *overfitting* or *high variance*

Alternatively, we might find the out of sample Performance Metric better than in-sample!

- this indicates that there is unrealized potential in the training data
- this is called *underfitting* or *high bias*

Overfitting is a problem because it means that Θ has been made overly sensitive to the training examples.

It is unlikely to generalize (i.e., predict out of sample) well.

Underfitting is a problem because it means your model may be able to be improved

Let's see how to deterimine whether we have an overfitting or underfitting problem.

We can quantify these concepts in terms of the relationship between the Training Error and the Validation Error.

High Variance

- if Training Error $<<$ Validation Error
    - overfit: poor generalization out of sample
        - model may be too complex relative to quantity of Training Data
            - simplify model (fewer parameters)
            - use regularization to reduce number of non-zero parameters
            - increase Training Data (e.g., data augmentation)

High Bias

- if Training Error $>$ Validation Error or Training Error seems "high"
    - underfit: sub-optimal performance in sample
        - try a more complex (more parameters) model
        - train model longer (assuming optimization via search, rather than closed form solution)
        - what is "high" ? Compare to a baseline model

Regularization: reducing overfitting

What can we do to combat over-fitting ?

Recall that LinearRegression model is optimizing a Loss Function $\mathcal{L}$ which was initially identical to the performance metric MSE in our example.

We will create a *weighted penalty* term

$$P = \alpha Q$$

where Q is a function of Θ in order to impose a cost on using too many degrees of freedom.

The new loss function becomes

$$\mathcal{L}' = \mathcal{L} + \alpha Q$$

The weight α balances the original Cost fuction with the penalty.

Observe that the new Loss function is **no longer identical** to the performance metric, which remains unchanged.

We'll show several modified Regression models and identify the penalty used.

Ridge Regression (external/PythonDataScienceHandbook/notebooks/05.06-Linear-Regression.ipynb#Ridge-regression-%28L_2-Regularization%29)

Penalty is: sum (over parameters) of squared parameter value

$$P = \alpha \sum_{n=1}^{N} \theta_n^2$$

Also known as L_2 regularizer.

This tends to push parameters θ_n towards smaller values.

Lasso Regression (external/PythonDataScienceHandbook/notebooks/05.06-Linear-Regression.ipynb#Lasso-regression-(L_1-regularization))

Penalty is: sum (over parameters) of parameter's absolute value

$$P = \alpha \sum_{n=1}^{N} |\theta_n|$$

Also know as the L_1 regularizer.

This tends to push parameters θ_n towards 0

TIP - Know your Cost Function ! - This is what is driving your model's fitting process. Does it make sense for your problem ? - Don't try to force a square peg into a round hole.

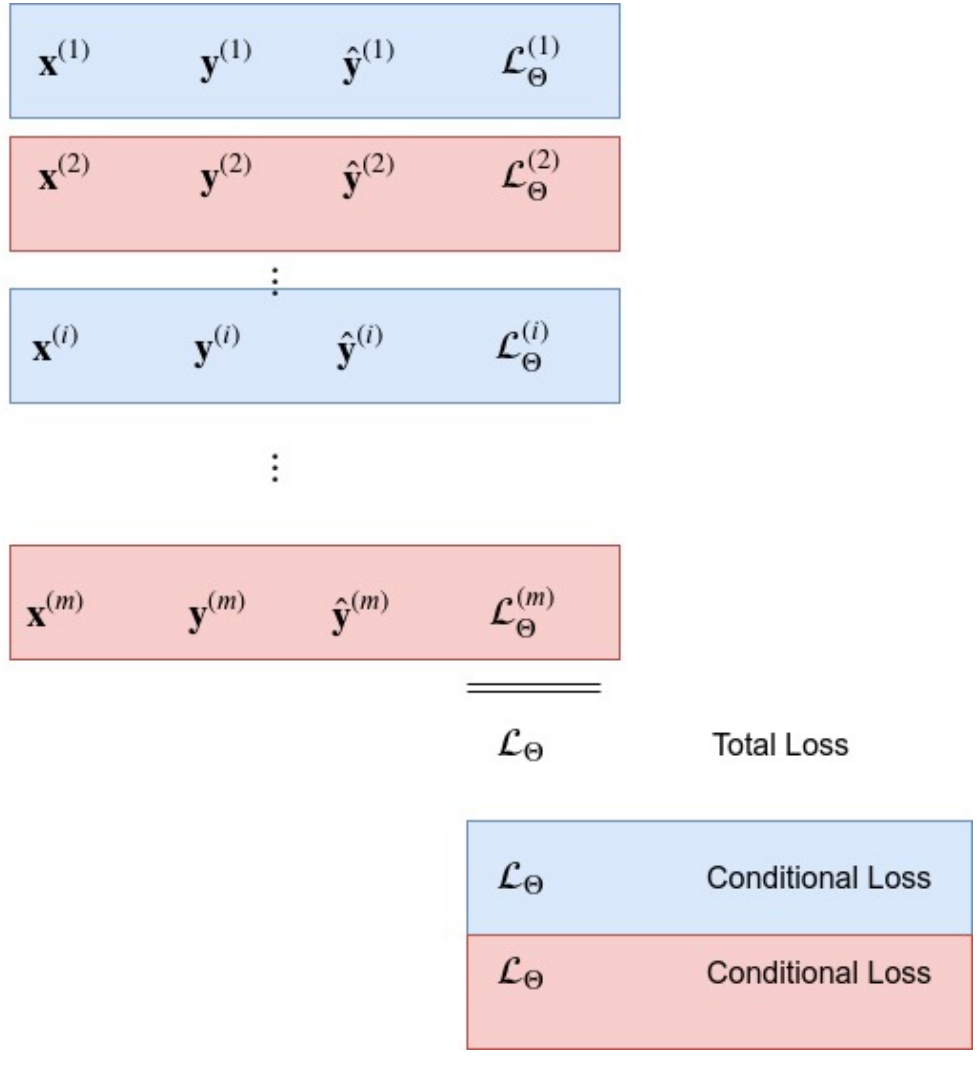
Error Analysis

Error analysis diagram should be on TEST and performance metric, vs train and loss

Conditional loss can be used to justify understanding loss for fitting

- e.g., imbalanced data focus optimizer on majority class

Error analysis: conditional loss



In [6]:
```
print("Done !")
```

```
Done !
```

Extra: try adding noise and using higher order polynomial to show danger of overfitting