Prepare data: transformations

Transforming data (Recipe C.3) may be **the most important** step of the multi-step Recipe !

Recipe for Machine Learning

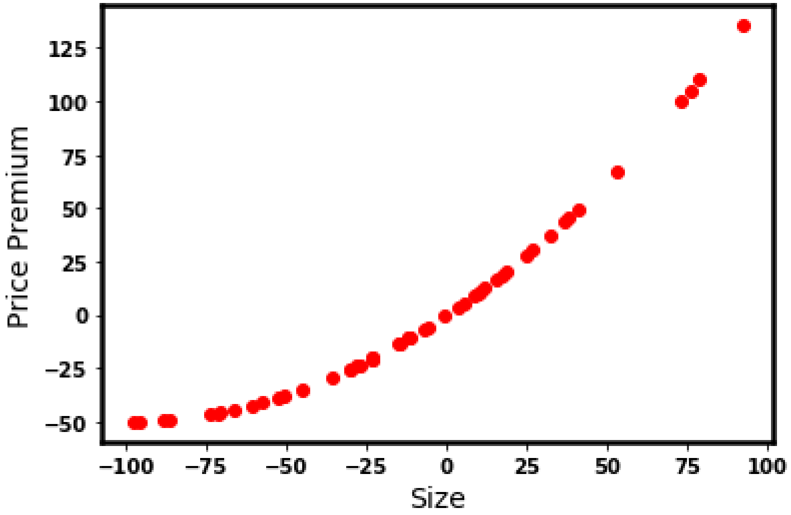
It is often the case that the "raw" features given to us don't suffice

- we may need to create "synthetic" features.
- This is called **feature engineering**.

Recall: our "curvy" data set from the previous lecture:

In [4]:

```
(xlabel, ylabel) = ("Size", "Price Premium")
v1, a1 = 1, .005
v2, a2 = v1, a1*2
curv = recipe_helper.Recipe_Helper(v = v2, a = a2)
X_curve, y_curve = curv.gen_data(num=50)
_= curv.gen_plot(X_curve,y_curve, xlabel, ylabel)
```

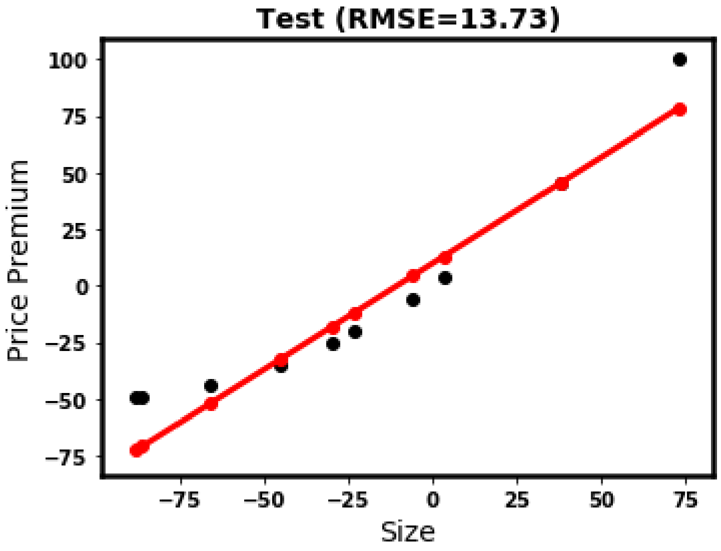


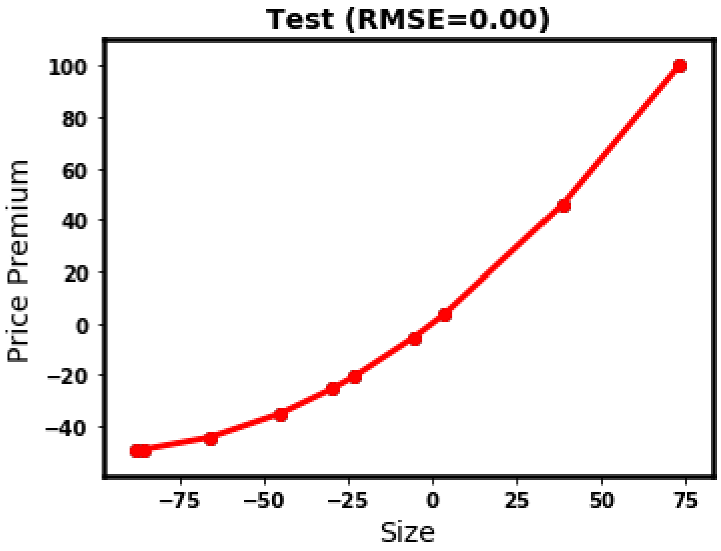
And compare the out of sample performance on this data set

- On a linear model (single, raw feature)
- On a model with a second feature (squared version of raw feature)

In [5]:

```
model_results = curv.compare_regress(X_curve, y_curve, xlabel=xlabel, ylabel=ylabel, visible=True, plot_train=False)
```





Adding the synthetic feature was key to better performance (lower RMSE).

Feature engineering, or transformations

- takes an example: vector $\mathbf{x}^{(\mathbf{i})}$ with n features
- produces a new vector $\tilde{\mathbf{x}}^{(\mathbf{i})}$, with n' features

We ultimately fit the model with the transformed *training* examples.

We can apply multiple transformations, each

- Adding new synthetic features
- Further transforming synthetic features

Feature Engineering

The above diagram shows multiple transformations

- organized as a sequence (sometimes called a *pipeline*) of independent transformations $T_1, T_2, \ldots, T_t$

$$
\begin{aligned}
\tilde{\mathbf{x}}_{(1)} &= T_1(\mathbf{x}) \\
\tilde{\mathbf{x}}_{(2)} &= T_2(\tilde{\mathbf{x}}_{(1)}) \\
\vdots \\
\tilde{\mathbf{x}}_{(l+1)} &= T_{(l+1)}(\tilde{\mathbf{x}}_{(l)})
\end{aligned}
$$

We write the final transformed $\tilde{\mathbf{x}}$ as a function T that is the composition of each transformation function

$$\tilde{\mathbf{x}} = T(\mathbf{x}) = T_t(\ T_{t-1}(\ldots T_1(\mathbf{x}) \ldots)\ )$$

The length of the final transformed vector $\tilde{\mathbf{x}}$ may differ from the n, the length of the input $\mathbf{x}$

- may add features
- may drop features

The predictions are now a function of $\tilde{\mathbf{x}}$ rather than $\mathbf{x}$

$$\hat{\mathbf{y}} = h_{\Theta}(\tilde{\mathbf{x}})$$

Example transformation: Missing data imputation

The first transformation we encountered added a feature ($\mathbf{x}^2$ term) that improved prediction.

Some transformations alter existing features rather than adding new ones.

Transformations in detail will be the subject of a separate lecture but let's cover the basics.

Let's consider a second reason for transformation: filling in (imputing) missing data for a feature.

#	$\mathbf{x}_1$	$\mathbf{x}_2$
1	1.0	10
2	2.0	20
⋮	⋮	⋮
i	2.0	NaN
⋮	⋮	⋮
m	. . .	

In the above: feature $\mathbf{x}_2$ is missing a value in example i: $\mathbf{x}_2^{\mathbf{(i)}} = \text{NaN}$

We will spend more time later discussing the various ways to deal with missing data imputation.

For now: let's adopt the common strategy of replacing it with the median of the defined values:

$$\text{median}(\mathbf{x}_2) = \text{median}(\{\mathbf{x}_2^{\mathbf{(i)}} | 1 \le i \le m, \mathbf{x}_2^{\mathbf{(i)}} \ne \text{NaN}\})$$

This imputation is a kind of data transformation: replacing an undefined value.

Without this transformation: the algorithm that implements our model

- May fail
- May impute a less desirable value, since it lacks specific knowledge of our problem

”Fitting” transformations

The behavior of our models for prediction have parameters Θ.

It might not be obvious that transformations have parameters $\Theta_{\text{transform}}$ as well

$$\tilde{\mathbf{x}} = T_{\Theta_{\text{transform}}}(\mathbf{x})$$

For example: when missing data imputation for a feature substitutes the mean/median feature value

- $\Theta_{\text{transform}}$ stores this value

We use the term "fitting" to describe the process of solving for $\Theta_{\text{transform}}$

- Unlike Θ, one doesn't usually find a "optimal" value for $\Theta_{\text{transform}}$

Our prediction is thus

$$
\begin{array}{lll}
\hat{\mathbf{y}} & = & h_\Theta(\tilde{\mathbf{x}}) \\
 & = & h_\Theta(\, T_{\Theta_{\text{transform}}}(\mathbf{x})\,)
\end{array}
$$

The process of Transformations is similar to fitting a model and predicting.

The parameters in $\Theta_\text{transform}$

- are "fit" by examining all training data $\mathbf{X}$
- once fit, we can transform ("predict") *any* example (whether it be training/validation or test)

Applying transformations consistently

Since the prediction is now

$$\hat{\mathbf{y}} \quad = \quad h_\Theta(\, \tilde{\mathbf{x}}\,) \qquad \text{where } \tilde{\mathbf{x}} = T_{\Theta_\text{transform}}(\mathbf{x})$$

each and every input $\mathbf{x}$ must be transformed

- Training examples
- Test examples

That is: the transformation is applied consistently across all examples, regardless of their source

If we didn't apply the same transformation to both training and test examples

- We would violate the Fundamental Assumption of Machine Learning

However

- $\Theta_{\text{transform}}$ is fit **only** to training examples
- It is **not** recalculated on a set of test examples

Here's the picture

Feature engineering: fit, then transform

There are several reasons not to re-fit on test examples

- It would be a kind of "cheating" to see all test examples (required to fit)
- You should assume that you only encounter one test example at a time, not as a group

Using pipelines to avoid cheating in cross validation

Although we start off with the best intentions, it is easy to accidentally "cheat"

- When we combine transformations and cross-validation (to measure out of sample performance)
- Is surprisingly common !

k-fold cross-validation:

- Divides the training examples into k "folds"
- A model is fit k times
- Each fit
    - Uses $(k-1)$ folds for training
    - The remaining fold is considered "out of sample" for that fit
- This gives us k Performance Metrics: a distribution of out of sample performance

Cross Validation/Test split

Consider the difference between fitting $\Theta_{\text{transform}}$

- Once, on *all* the training examples, *before* applying cross-validation
- Separately for each of the k fits of Cross-Validation
    - Using the $(k-1)$ folds used for training in this fit

For example, when Fold_k is out of sample

$$\Theta_{\mathrm{transform}} = f([\mathrm{Fold}_1, \mathrm{Fold}_2, \ldots \mathrm{Fold}_{k-1}, \mathrm{Fold}_k])$$

$$\text{versus}$$

$$\Theta_{\mathrm{transform}} = f([\mathrm{Fold}_1, \mathrm{Fold}_2, \ldots \mathrm{Fold}_{k-1})$$

In the first case, we are cheating !

- Fold k is out of sample for this fit
- And should **not** influence $\Theta_\text{transform}$

The second case avoids this problem

- With seemingly a lot more work
- Fitting $\Theta_\text{transform}$ multiple times

Perhaps the increase in effort is one reason this subtle cheating is overlooked.

Fortunately, a good toolkit for ML (e.g., `sklearn`) can facilitate proper transformation fitting without extra effort.

Let's explore Transformation pipelines in `sklearn` (Transformations_Pipelines.ipynb)

We will see this in action within the notebook for Classification.

In [6]:

```
print("Done")
```

```
Done
```