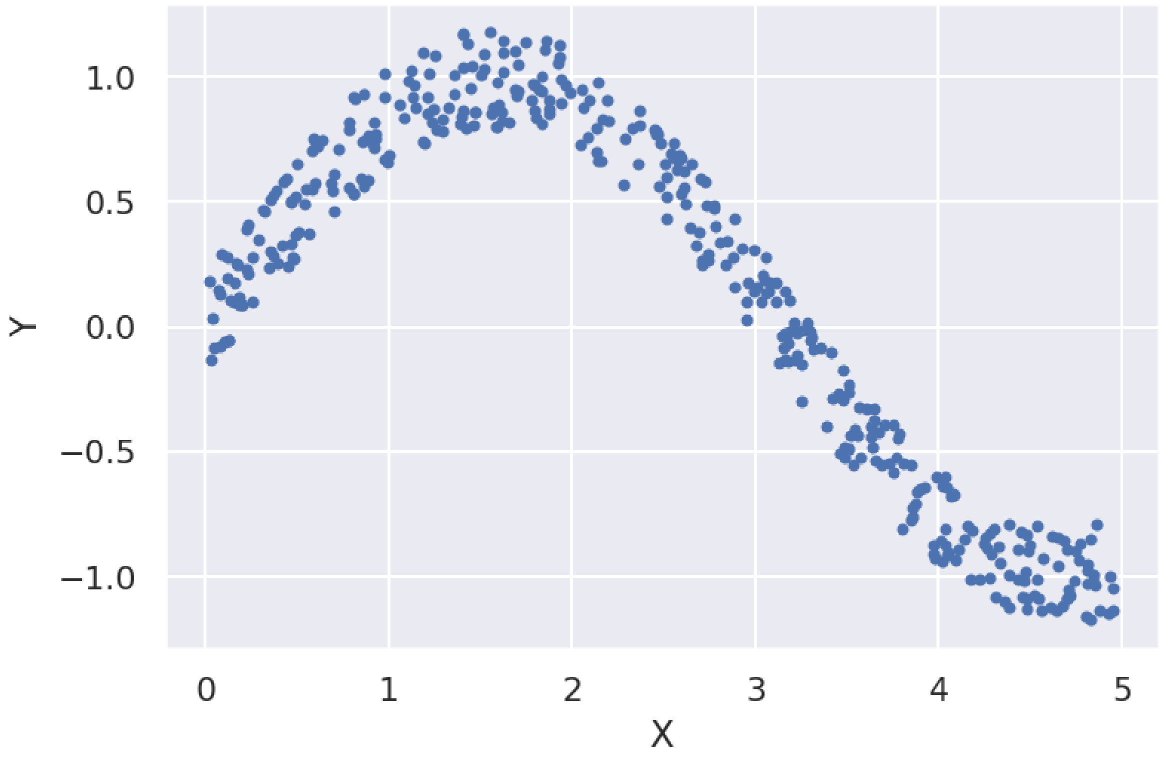
In [47]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

Bias, Variance, and Overfitting

In this notebook, we want to illustrate the notion of overfitting. We'll generate data and use a decision tree to illustrate an example of the bias-variance tradeoff. Let's start by generating and plotting some data...

In [48]:
```
from IPython import display
from sklearn.externals.six import StringIO
import numpy as np
from matplotlib import pylab as plt

# Create a random dataset
rng = np.random.RandomState(42) # fix the seed so the result are stable
N_points = 400
X = np.sort(5 * rng.rand(N_points, 1), axis=0)
y = np.sin(X).ravel() + .4 * (0.5 - rng.rand(N_points))

plt.plot(X, y, 'b.')
plt.xlabel('Data')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Decision Trees

A Decision Tree is a recursive binary tree structure. You can read more about decision trees [here (http://scikit-learn.org/stable/modules/tree.html)](http://scikit-learn.org/stable/modules/tree.html).

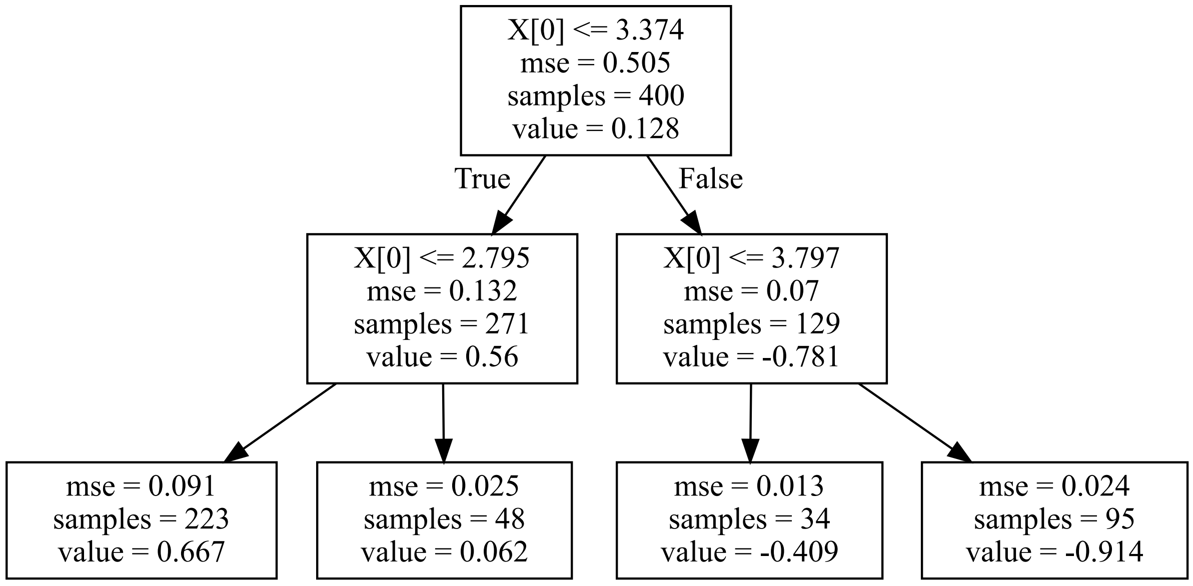
In [49]:
```
# Train a 2 level decision tree

from ipywidgets import interactive, IntSlider
from sklearn import tree

def train_and_plot(max_depth):
    est = tree.DecisionTreeRegressor(max_depth=max_depth)
    est.fit(X, y)

    plt.plot(X, y, 'b.', label='data')
    line = plt.plot(X, est.predict(X), 'r-', label='model')
    plt.setp(line, linewidth=3.)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend(loc='upper right');
    plt.show()

max_depth_slider=IntSlider(min=1,max=8,step=1,value=2)
interactive(train_and_plot, max_depth=max_depth_slider)
```

Exercise: Try playing around with `max_depth` in the above above cell. What happens when you set it to `max_depth = 2`. Does the fitted model look better or worse? What about `4` or `8`? We can tell (visually) that `max_depth = 2` is *underfitting* and `max_depth = 8` is *overfitting*. But how can we do this computationally?

We can visualize the tree that was trained for different values of `max_depth` using `graphviz`.

In [50]:
```
import graphviz

est = tree.DecisionTreeRegressor(max_depth=2)
est.fit(X, y)
graphviz.Source(tree.export_graphviz(est, out_file=None))
```

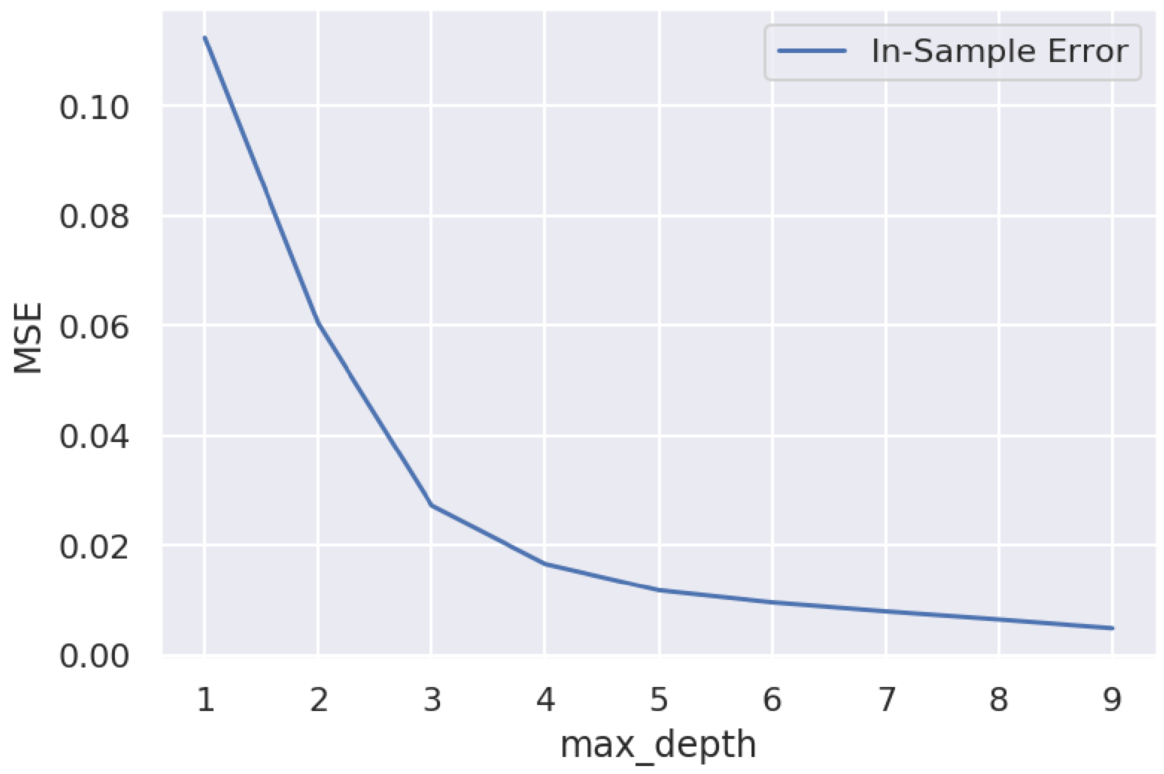
Out[50]:



In-sample error

Naively, you might think that we could just measure the error of the model and choose the model with the best error. For example, let's define the error as the mean squared error (MSE). Let's try that below

In [51]:
```
from sklearn import metrics

max_depths = range(1, 10)
in_sample_errors = []
for max_depth in max_depths:
    # specifiying min_samples_leaf can reduce overfitting
    y_pred = tree.DecisionTreeRegressor(max_depth=max_depth).fit(X, y).predict
(X)
    in_sample_errors.append(metrics.mean_squared_error(y, y_pred))

plt.plot(max_depths, in_sample_errors, label='In-Sample Error')
plt.xlabel('max_depth')
plt.ylabel('MSE')
plt.legend(loc='upper right');
```



We can see that the error is decreasing with the depth of the tree. Our visual inspection told us models with `max_depth = 8` completely over-fit the error. It turns out that the *In-Sample Error* that we calculated above will always decrease with the complexity of the model (in this case, the depth of the tree). We can see from above that this leads us to *overfit* the data. In order to test how well our model *generalizes*, we need to see how it performs on new data.

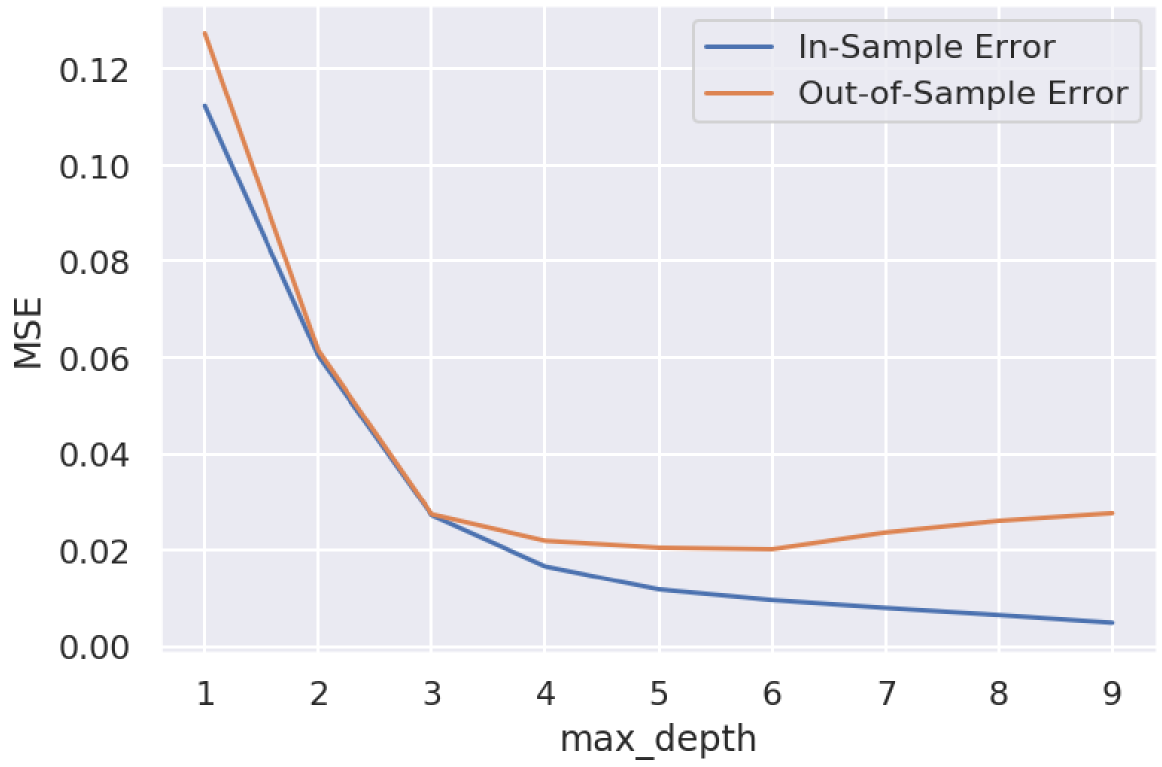
Out-of-sample error

One way to do this is to (randomly) split the data into training and test sets. We train on the training set and test the resulting model on the test set. Since the trained model never saw the test data, we can evaluate the performance on the test data.

In [52]:
```
from sklearn import model_selection
from sklearn import metrics

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test
_size=0.2, random_state=42)
X.shape, X_train.shape, X_test.shape
```

Out[52]:
```
((400, 1), (320, 1), (80, 1))
```

In [53]:
```
test_errors = []
for max_depth in max_depths:
    est = tree.DecisionTreeRegressor(max_depth=max_depth).fit(X_train, y_train
)
    y_pred = est.predict(X_test)
    test_errors.append(metrics.mean_squared_error(y_test, y_pred))

plt.plot(max_depths, in_sample_errors, label='In-Sample Error')
plt.plot(max_depths, test_errors, label='Out-of-Sample Error')
plt.xlabel('max_depth')
plt.ylabel('MSE')
plt.legend(loc='upper right');
```



Variance-bias tradeoff

This is the picture we were looking for! The *In-Sample Error* is always decreasing but the out *Out-of-Sample Error* initially decreases with model complexity (higher `max_depth`) but ultimately increases again. The *Out-of-Sample Error* is the metric to look at when evaluating overfitting.

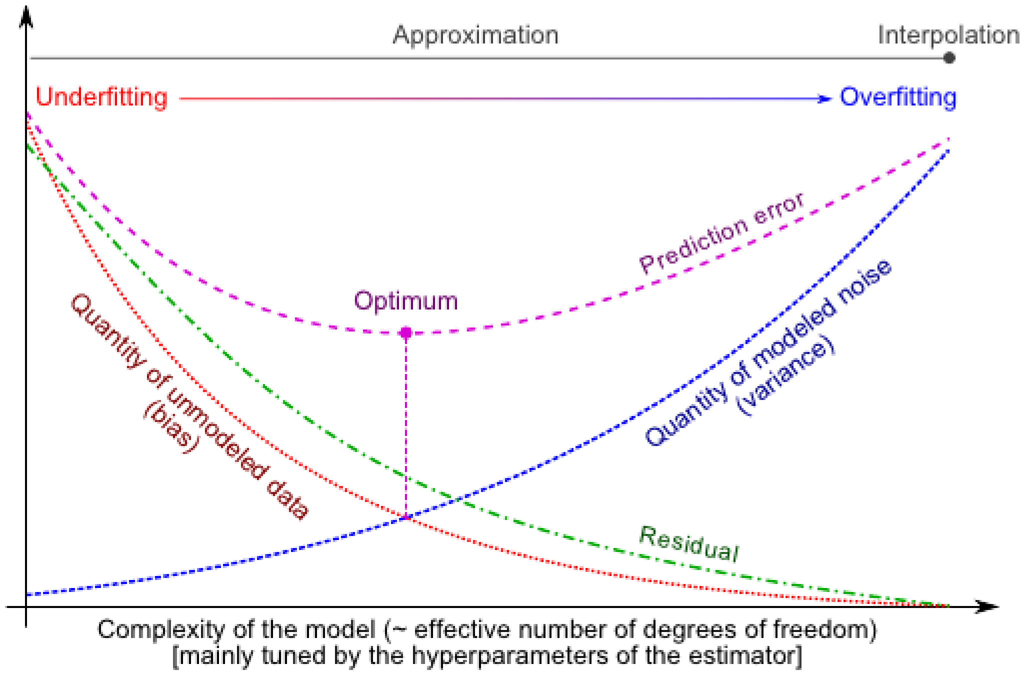
It turns out that this is an illustration of a very general problem in machine-learning called **Bias-Variance tradeoff** (the concept is so general that it even has a Wikipedia article (https://en.wikipedia.org/wiki/Bias%E2%80%93variance_dilemma). A more rigorous account can be found here (http://www.brnt.eu/phd/node14.html)). The tradeoff tells us that we can decompose our *Out-of-Sample Error* into

$$\text{Out-of-Sample Error} = \text{Bias} + \text{Variance}.$$

The *Bias* corresponds to how far off we expect the model to deviate from reality (i.e. the model's bias) because of parametric assumptions (e.g. we forced the model to be linear or to be a tree of maximum depth 2). It is given by the *In-Sample Error* of the above plot and always goes down with complexity. High Bias models correspond to *underfitting*.

The *Variance* accounts for the fact that the model was only trained on a (noisy) subset of the data and that the idiosyncratic noise in the data is therefore likely to contribute some variance to the model. The more complex we allow the model to be, the more likely we are to overfit by picking up more of this noise. High variance models correspond to *overfitting*.

We can also think of bias as unmodeled data and variance as modeled noise. As we increase the complexity of the model, we will necessarily model more of the data (reduce bias, reduce underfitting) but also start modeling noise (increase variance, increase overfitting). Here's a helpful diagram of the decomposition. Notice that at the optimal point, we have not yet learned on all our signal (still unmodeled data left) and we have picked up some noise and overfitting.



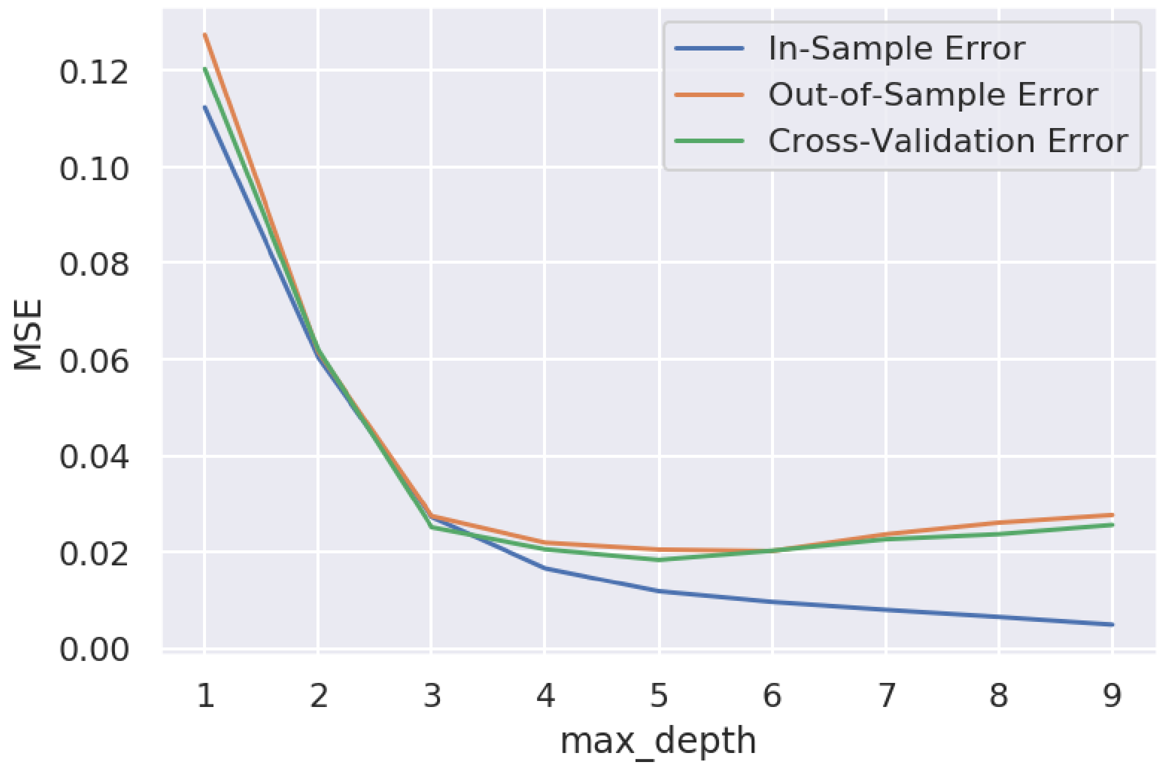
Cross-validation strategies

The strategy we used above was a fairly elementary. We could, for example, split the data into k parts (called *folds*), train on $k-1$ of them and test the resulting model on the last one. This is called *K-Fold Validation*. There are many variations. Here are a few:

- *Stratified K-Fold Validation*: restrict the folds to have the same percentage of y's as the full sample.
- *Leave-One-Out*: If $k=n$ K-Fold Validation.
- *Leave-p-Out*: You can probably guess ...

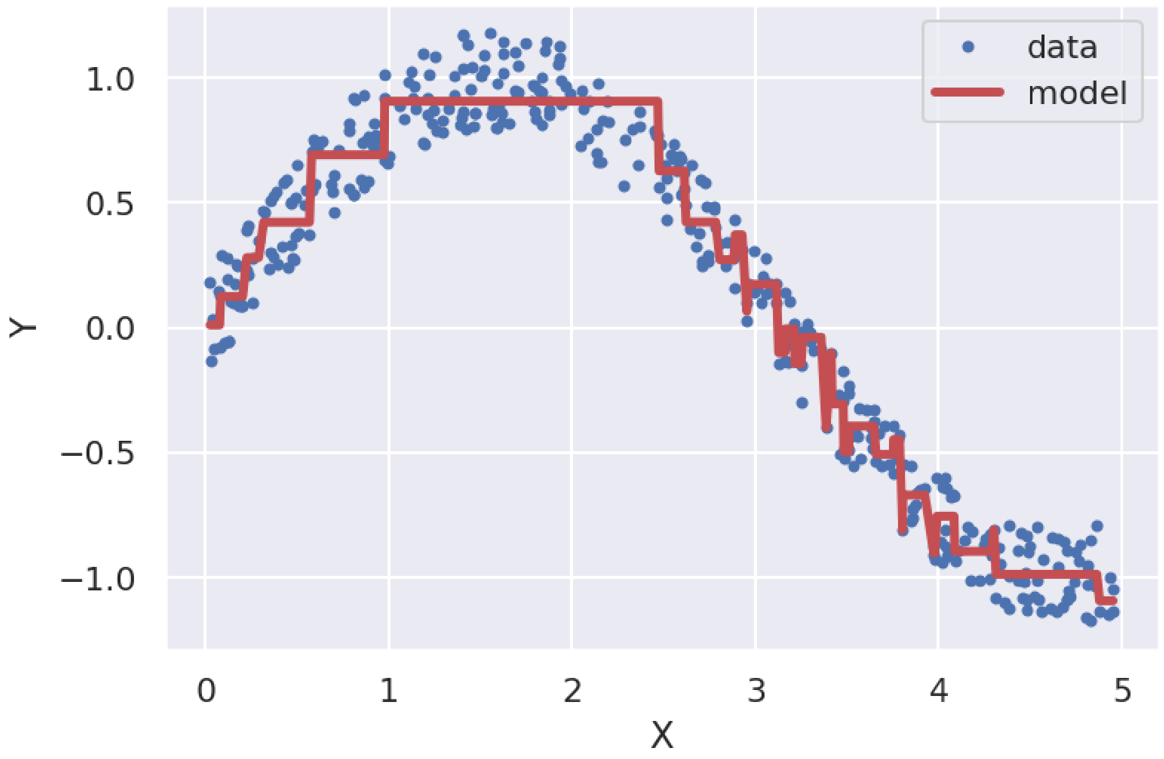
For documentation on how to use these in Scikit-Learn, take a look at this page (http://scikit-learn.org/stable/modules/cross_validation.html).

In [54]:
```
cv_test_errors = []

# The K-fold Cross Validation algorithm has a bug (feature?) where it makes th
e
# folds consecutive elements.  Because the input data is presorted along X, we
have
# to first shuffle the input data.  (Exercise: What happens if we don't?)
indices = np.random.permutation(range(len(y)))
X_random_order, y_random_order = X[indices], y[indices]

for max_depth in max_depths:
    est = tree.DecisionTreeRegressor(max_depth=max_depth)

    # Note that cross_val_score performs K-fold Cross Validation but returns a
score,
    # which is the negative of the error.
    cv_test_error = -model_selection.cross_val_score(
        est,
        X_random_order,
        y_random_order,
        cv=5,  # Number of folds
        scoring='neg_mean_squared_error'
    )
    cv_test_errors.append(cv_test_error.mean())

plt.plot(max_depths, in_sample_errors, label='In-Sample Error')
plt.plot(max_depths, test_errors, label='Out-of-Sample Error')
plt.plot(max_depths, cv_test_errors, label='Cross-Validation Error')
plt.xlabel('max_depth')
plt.ylabel('MSE')
plt.legend(loc='upper right');
```



Once the best hyperparameters are found, the model should be trained on the full data set.

```
In [55]: best_est = tree.DecisionTreeRegressor(max_depth=max_depths[np.argmin(cv_test_e
         rrors)])
         best_est.fit(X, y)

         plt.plot(X, y, 'b.', label='data')
         line = plt.plot(X, best_est.predict(X), 'r-', label='model')
         plt.setp(line, linewidth=3.)
         plt.xlabel('X')
         plt.ylabel('Y')
         plt.legend(loc='upper right')
         plt.show()
```

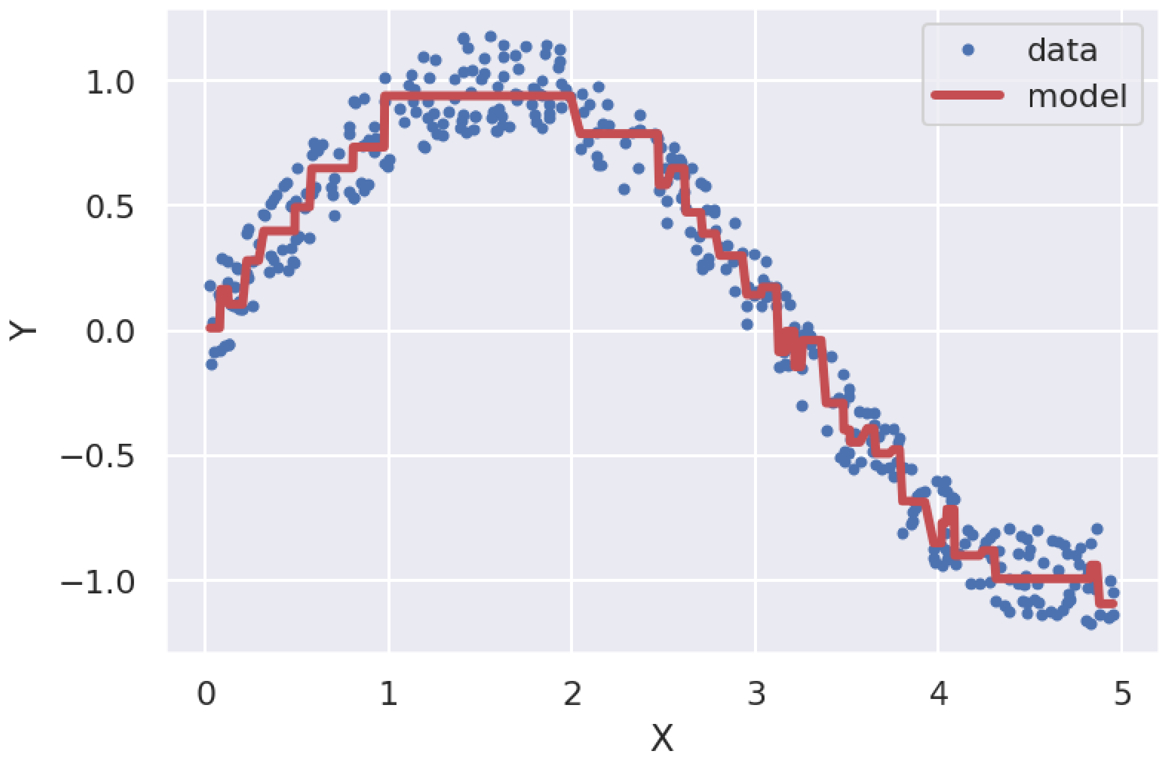


Grid search for tuning hyperparameters

This process is automated by the `GridSearchCV` estimator. It takes an estimator and the hyperparameter ranges to test. Inside the `.fit` method, cross-validation will be done to score each set of hyperparameters, and the best model will be selected and fit with the full data. After fitting, it has `.best_estimator_`, `.best_params_` and `.best_score_` attributes set, and the `.predict` method will call the best estimator.

In [56]:
```
from sklearn import model_selection

est = tree.DecisionTreeRegressor()
gs = model_selection.GridSearchCV(
    est,
    {"max_depth": range(1,11),
    "min_samples_leaf": range(1,10)},
    cv=5,  # 5-fold cross validation
    n_jobs=2,  # run each hyperparameter in one of two parallel jobs
    scoring='neg_mean_squared_error'
)
gs.fit(X_random_order, y_random_order)
print (gs.best_params_)

plt.plot(X, y, 'b.', label='data')
line = plt.plot(X, gs.predict(X), 'r-', label='model')
plt.setp(line, linewidth=3.)
plt.xlabel('X')
plt.ylabel('Y')
plt.legend(loc='upper right')
plt.show()
```

```
{'max_depth': 6, 'min_samples_leaf': 5}
```



In [61]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

pipe = Pipeline([
    ('scale', StandardScaler()),
    ('tree', tree.DecisionTreeRegressor())
])

gs_pipe = model_selection.GridSearchCV(
    pipe,
    {'tree__min_samples_leaf' : range(1,11)},
    cv = 5,
    n_jobs = 2,
    scoring = 'neg_mean_squared_error'
).fit(X_random_order, y_random_order)

gs_pipe.best_estimator_.named_steps['tree']
```

Out[61]:
```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=10,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=None, splitter='best')
```

In [63]:
```
from sklearn.linear_model import RidgeCV
RidgeCV?
```

Questions

1. Cross validation works well in many situations. Can you think of any situations where it is not straightforward to apply? Is there a natural grouping where you would expect intragroup variation to be smaller than intergroup variation?

Exercises

1. According to the above picture, choosing the minimum *Out-of-Sample Error* or (total error) gives a model with `max_depth` of approximately 5. Plot the model and see if the model seems reasonable.
2. There are other ways to constrain the growth of trees besides `max_depth`. For example `min_samples_split` constrains the minimum samples below which the tree will not split and `min_samples_leaf` constrains the minimum number of samples in a leaf (terminal node of a tree). Read about them here (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html). Does increasing these increase or decrease model complexity? Use cross validation to compute the optimal set of `min_samples_split`, `min_samples_leaf`, and `max_depth`. *Hint*: you should use the `grid_search` function (http://scikit-learn.org/stable/modules/grid_search.html) or `GridSearchCV` to find the optimal combination.
3. Finally, we used `model_selection.cross_val_score` to compute a cross validation score inside a `for` loop. The MSE for `cross_val_score` and `grid_search` should align exactly. Why do these numbers align but the ones from `train_test_split` and `cross_val_score` do not?

Exit Tickets

1. Explain the variance-bias tradeoff to your grandmother.
2. What are the benefits/drawbacks to using cross-validation versus leaving out an untouched cross-validation set?
3. In the four scenarios where in-sample error is high/low and out-of-sample error is high/low, describe what is happening in words.

Spoilers

Answers

Here are two examples when cross-validation might fail:

- For example, in a modeling problem, men and women may behave very differently. Cross validation would only tell you that you have an OK model for each.
- With the stock market, a model that did well in 2000 at the height of the tech bubble, might not have done well in 2001 as the market was collapsing.

Exit Ticket Answers

1. You can think of the bias-variance tradeoff as a tradeoff between having a flexible or noisy model.
2. Cross validation lets you use all data while determining best hyperparameters, but it doesn't leave a completely untouched data set for evaluation purposes.
3. High in-sample, high out-of-sample: underfit model; Low in-sample, high out-of-sample: overfit model; Low in-sample, low out-of-sample: well-fit model