Machine Learning Engineer Nanodegree

Model Evaluation & Validation

Project 1: Predicting Boston Housing Prices

Welcome to the first project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been written. You will need to implement additional functionality to successfully answer all of the questions for this project. Unless it is requested, do not modify any of the code that has already been included. In this template code, there are four sections which you must complete to successfully produce a prediction with your model. Each section where you will write code is preceded by a **STEP X** header with comments describing what must be done. Please read the instructions carefully!

In addition to implementing code, there will be questions that you must answer that relate to the project and your implementation. Each section where you will answer a question is preceded by a **QUESTION X** header. Be sure that you have carefully read each question and provide thorough answers in the text boxes that begin with "**Answer:**". Your project submission will be evaluated based on your answers to each of the questions.

A description of the dataset can be found here (https://archive.ics.uci.edu/ml/datasets/Housing), which is provided by the **UCI Machine Learning Repository**.

Getting Started

To familiarize yourself with an iPython Notebook, **try double clicking on this cell**. You will notice that the text changes so that all the formatting is removed. This allows you to make edits to the block of text you see here. This block of text (and mostly anything that's not code) is written using Markdown (http://daringfireball.net/projects/markdown/syntax), which is a way to format text using headers, links, italics, and many other options! Whether you're editing a Markdown text block or a code block (like the one below), you can use the keyboard shortcut **Shift + Enter** or **Shift + Return** to execute the code or text block. In this case, it will show the formatted text.

Let's start by setting up some code we will need to get the rest of the project up and running. Use the keyboard shortcut mentioned above on the following code block to execute it. Alternatively, depending on your iPython Notebook program, you can press the **Play** button in the hotbar. You'll know the code block executes successfully if the message *"Boston Housing dataset loaded successfully!"* is printed.

In [51]:
```
# Importing a few necessary libraries
import numpy as np
import matplotlib.pyplot as pl
from sklearn import datasets
from sklearn.tree import DecisionTreeRegressor

# Make matplotlib show our plots inline (nicely formatted in the notebook)
%matplotlib inline

# Create our client's feature set for which we will be predicting a selling price
CLIENT_FEATURES = [[11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385, 24, 680.0, 20.20, 332.09, 12.13]]
CLIENT_FEATURES = np.array(CLIENT_FEATURES)
CLIENT_FEATURES = CLIENT_FEATURES.reshape(1, -1)

# Load the Boston Housing dataset into the city_data variable
city_data = datasets.load_boston()

# Initialize the housing prices and housing features
housing_prices = city_data.target
housing_features = city_data.data

print "Boston Housing dataset loaded successfully!"
```

```
Boston Housing dataset loaded successfully!
```

Statistical Analysis and Data Exploration

In this first section of the project, you will quickly investigate a few basic statistics about the dataset you are working with. In addition, you'll look at the client's feature set in `CLIENT_FEATURES` and see how this particular sample relates to the features of the dataset. Familiarizing yourself with the data through an explorative process is a fundamental practice to help you better understand your results.

Step 1

In the code block below, use the imported `numpy` library to calculate the requested statistics. You will need to replace each `None` you find with the appropriate `numpy` coding for the proper statistic to be printed. Be sure to execute the code block each time to test if your implementation is working successfully. The print statements will show the statistics you calculate!

In [52]:
```
# Number of houses in the dataset
total_houses = housing_features.shape[0]

# Number of features in the dataset
total_features = housing_features.shape[1]

# Minimum housing value in the dataset
minimum_price = np.min(housing_prices)

# Maximum housing value in the dataset
maximum_price = np.max(housing_prices)

# Mean house value of the dataset
mean_price = np.mean(housing_prices)

# Median house value of the dataset
median_price = np.median(housing_prices)

# Standard deviation of housing values of the dataset
std_dev = np.std(housing_prices)

# Show the calculated statistics
print "Boston Housing dataset statistics (in $1000's):\n"
print "Total number of houses:", total_houses
print "Total number of features:", total_features
print "Minimum house price:", minimum_price
print "Maximum house price:", maximum_price
print "Mean house price: {0:.3f}".format(mean_price)
print "Median house price:", median_price
print "Standard deviation of house price: {0:.3f}".format(std_dev)
```

```
Boston Housing dataset statistics (in $1000's):

Total number of houses: 506
Total number of features: 13
Minimum house price: 5.0
Maximum house price: 50.0
Mean house price: 22.533
Median house price: 21.2
Standard deviation of house price: 9.188
```

Question 1

As a reminder, you can view a description of the Boston Housing dataset here (https://archive.ics.uci.edu/ml/datasets/Housing), where you can find the different features under **Attribute Information**. The `MEDV` attribute relates to the values stored in our `housing_prices` variable, so we do not consider that a feature of the data.

Of the features available for each data point, choose three that you feel are significant and give a brief description for each of what they measure.

Remember, you can **double click the text box below** to add your answer!

Answer: The three features that I am choosing are: 1) CRIM - The per capita crime rate by town. This statistic measures the average number of crimes committed over the number of households, per town. The fact that the statistic is per capita means that the statistic is not skewed when, for example, a high number of crimes are located where there are a large number of people.
2) RM: average number of rooms per dwelling. This measures, as described, the average number of rooms per house in the surrounding area. I believe that this statistic, more than any others, reflects the value of the homes surrounding the client home, and therefore describes what "comparable" homes are selling for. Researching prices of comparable homes is a very common method of appraisal. 3) PTRATIO: pupil-teacher ratio by town. This statistic is the measure of teachers per students. More teachers per students indicates better, more well funded schools, while less teachers is an indication of overcrowded and typically lower income demographics.

Question 2

Using your client's feature set `CLIENT_FEATURES`, which values correspond with the features you've chosen above?
Hint: Run the code block below to see the client's data.

In [53]:
```
print CLIENT_FEATURES
```

```
[[  1.19500000e+01   0.00000000e+00   1.81000000e+01   0.00000000e+00
    6.59000000e-01   5.60900000e+00   9.00000000e+01   1.38500000e+00
    2.40000000e+01   6.80000000e+02   2.02000000e+01   3.32090000e+02
    1.21300000e+01]]
```

Answer: The value that corresponds to the crime rate is 11.95 The value that corresponds to average number of rooms is 5.609. The number that corresponds to the number of students per teacher is 20.2.

Evaluating Model Performance

In this second section of the project, you will begin to develop the tools necessary for a model to make a prediction. Being able to accurately evaluate each model's performance through the use of these tools helps to greatly reinforce the confidence in your predictions.

Step 2

In the code block below, you will need to implement code so that the `shuffle_split_data` function does the following:

- Randomly shuffle the input data `X` and target labels (housing values) `y`.
- Split the data into training and testing subsets, holding 30% of the data for testing.

If you use any functions not already acessible from the imported libraries above, remember to include your import statement below as well!
Ensure that you have executed the code block once you are done. You'll know if the `shuffle_split_data` function is working if the statement *"Successfully shuffled and split the data!"* is printed.

In [59]:
```
from sklearn import datasets
from sklearn import cross_validation
from sklearn.svm import SVC

def shuffle_split_data(X, y):
    """ Shuffles and splits data into 70% training and 30% testing subse
ts,
        then returns the training and testing subsets. """

    # Shuffle and split the data
    X_train, X_test, y_train, y_test = cross_validation.train_test_split
(housing_features, housing_prices, test_size=0.3, random_state=0)

    # Return the training and testing data subsets
    return X_train, y_train, X_test, y_test


# Test shuffle_split_data
try:
    X_train, y_train, X_test, y_test = shuffle_split_data(housing_featur
es, housing_prices)
    print "Successfully shuffled and split the data!"
except:
    print "Something went wrong with shuffling and splitting the data."
```

```
Successfully shuffled and split the data!
```

Question 4

Why do we split the data into training and testing subsets for our model?

Answer: We split the data into training and testing subsets because the model requires both for us to be able to evaluate if it is effective in regards to the goal of predicting home prices. All models need to be both trained and tested. The training data is what we use to create an accurate model, and the testing data is what we use to test the accuracy of the model created by using the training data.

Step 3

In the code block below, you will need to implement code so that the `performance_metric` function does the following:

- Perform a total error calculation between the true values of the `y` labels `y_true` and the predicted values of the `y` labels `y_predict`.

You will need to first choose an appropriate performance metric for this problem. See the sklearn metrics documentation (http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics) to view a list of available metric functions. **Hint:** Look at the question below to see a list of the metrics that were covered in the supporting course for this project.

Once you have determined which metric you will use, remember to include the necessary import statement as well!
Ensure that you have executed the code block once you are done. You'll know if the `performance_metric` function is working if the statement *"Successfully performed a metric calculation!"* is printed.

In [55]:
```
from sklearn.metrics import mean_squared_error

def performance_metric(y_true, y_predict):
    """ Calculates and returns the total error between true and predicte
d values
        based on a performance metric chosen by the student. """

    error = mean_squared_error(y_true, y_predict)
    return error


# Test performance_metric
try:
    total_error = performance_metric(y_train, y_train)
    print "Successfully performed a metric calculation!"
except:
    print "Something went wrong with performing a metric calculation."
```

```
Successfully performed a metric calculation!
```

Question 4

Which performance metric below did you find was most appropriate for predicting housing prices and analyzing the total error. Why?

- *Accuracy*
- *Precision*
- *Recall*
- *F1 Score*
- *Mean Squared Error (MSE)*
- *Mean Absolute Error (MAE)*

Answer: The best choice is Mean Squared Error. First we need to decide if the data we are using makes classification or regression types of predictions. Since housing prices are continuous data, we are making regression types of predictions. Accuracy, precision, and recall are all used when predicting classification. F1 score is a weighted average of precision and recall, and is also used with classification data. Therefore either choosing MSE or MAE is most appropriate when dealing with continuous data and making regression type predictions. The mean square error has advantages over Mean Absolute Error including automatically converting negative numbers, and emphasizing larger errors.

Step 4 (Final Step)

In the code block below, you will need to implement code so that the `fit_model` function does the following:

- Create a scoring function using the same performance metric as in **Step 2**. See the sklearn `make_scorer` documentation (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).
- Build a GridSearchCV object using `regressor`, `parameters`, and `scoring_function`. See the sklearn documentation on GridSearchCV (http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html).

When building the scoring function and GridSearchCV object, *be sure that you read the parameters documentation thoroughly.* It is not always the case that a default parameter for a function is the appropriate setting for the problem you are working on.

Since you are using `sklearn` functions, remember to include the necessary import statements below as well!
Ensure that you have executed the code block once you are done. You'll know if the `fit_model` function is working if the statement *"Successfully fit a model to the data!"* is printed.

In [56]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import make_scorer
from sklearn.grid_search import GridSearchCV

def fit_model(X, y):
    """ Tunes a decision tree regressor model using GridSearchCV on the
input data X
        and target labels y and returns this optimal model. """

    # Create a decision tree regressor object
    regressor = DecisionTreeRegressor()

    # Set up the parameters we wish to tune
    parameters = {'max_depth':(1,2,3,4,5,6,7,8,9,10)}

    # Make an appropriate scoring function
    scoring_function = make_scorer(performance_metric, greater_is_better
=False)

    # Make the GridSearchCV object
    reg = GridSearchCV(regressor, parameters, scoring = scoring_functio
n, verbose = True)

    # Fit the learner to the data to obtain the optimal model with tuned
parameters
    reg.fit(X, y)

    # Return the optimal model
    return reg


# Test fit_model on entire dataset
try:
    reg = fit_model(housing_features, housing_prices)
    print "Successfully fit a model!"
except:
    print "Something went wrong with fitting a model."
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
Successfully fit a model!
```

```
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:    0.2s finished
```

Question 5

What is the grid search algorithm and when is it applicable?

Answer: The grid search algorithm is an algorithm that is used to systematically process multiple combinations of paramaters. One will choose which paramaters that one would like to test and the algorithm cross-validates (defined in the following question) each paramater combination and ultimately determines the combination that gives the best performance.

Question 6

What is cross-validation, and how is it performed on a model? Why would cross-validation be helpful when using grid search?

Answer: Cross-validation is a method that can be used to validate a model by assessing how the results of a statistical analysis will generalize to an independent data set. This is accomplished by dividing the available data into different partitions known as the training set and the testing set. The training set is used to teach or train models that best fit the data for purposes like prediction. The predicitive model is then tested on the data from the testing set to verify the performance of the model. This is usually done multiple times and then averaged in order to reduce variability. Cross-validation is helpful when performing grid search because it can be used to fine tune the paramaters. Where one alternative to using cross-validation would be a simple guess and check, that would be very time consuming, and can be accomplished more efficiently by using cross-validation.

Checkpoint!

You have now successfully completed your last code implementation section. Pat yourself on the back! All of your functions written above will be executed in the remaining sections below, and questions will be asked about various results for you to analyze. To prepare the **Analysis** and **Prediction** sections, you will need to intialize the two functions below. Remember, there's no need to implement any more code, so sit back and execute the code blocks! Some code comments are provided if you find yourself interested in the functionality.

In [60]:
```
def learning_curves(X_train, y_train, X_test, y_test):
    """ Calculates the performance of several models with varying sizes of training data.
        The learning and testing error rates for each model are then plotted. """

    print "Creating learning curve graphs for max_depths of 1, 3, 6, and 10. . ."

    # Create the figure window
    fig = pl.figure(figsize=(10,8))

    # We will vary the training set size so that we have 50 different sizes
    sizes = np.round(np.linspace(1, len(X_train), 50))
    train_err = np.zeros(len(sizes))
    test_err = np.zeros(len(sizes))

    # Create four different models based on max_depth
    for k, depth in enumerate([1,3,6,10]):

        for i, s in enumerate(sizes):

            # Setup a decision tree regressor so that it learns a tree with max_depth = depth
            regressor = DecisionTreeRegressor(max_depth = depth)

            # Fit the learner to the training data
            regressor.fit(X_train[:s], y_train[:s])

            # Find the performance on the training set
            train_err[i] = performance_metric(y_train[:s], regressor.predict(X_train[:s]))

            # Find the performance on the testing set
            test_err[i] = performance_metric(y_test, regressor.predict(X_test))

        # Subplot the learning curve graph
        ax = fig.add_subplot(2, 2, k+1)
        ax.plot(sizes, test_err, lw = 2, label = 'Testing Error')
        ax.plot(sizes, train_err, lw = 2, label = 'Training Error')
        ax.legend()
        ax.set_title('max_depth = %s'%(depth))
        ax.set_xlabel('Number of Data Points in Training Set')
        ax.set_ylabel('Total Error')
        ax.set_xlim([0, len(X_train)])

    # Visual aesthetics
    fig.suptitle('Decision Tree Regressor Learning Performances', fontsize=18, y=1.03)
    fig.tight_layout()
    fig.show()
```

In [47]:
```
def model_complexity(X_train, y_train, X_test, y_test):
    """ Calculates the performance of the model as model complexity incr
eases.
        The learning and testing errors rates are then plotted. """

    print "Creating a model complexity graph. . . "

    # We will vary the max_depth of a decision tree model from 1 to 14
    max_depth = np.arange(1, 14)
    train_err = np.zeros(len(max_depth))
    test_err = np.zeros(len(max_depth))

    for i, d in enumerate(max_depth):
        # Setup a Decision Tree Regressor so that it learns a tree with
depth d
        regressor = DecisionTreeRegressor(max_depth = d)

        # Fit the learner to the training data
        regressor.fit(X_train, y_train)

        # Find the performance on the training set
        train_err[i] = performance_metric(y_train, regressor.predict(X_t
rain))

        # Find the performance on the testing set
        test_err[i] = performance_metric(y_test, regressor.predict(X_tes
t))

    # Plot the model complexity graph
    pl.figure(figsize=(7, 5))
    pl.title('Decision Tree Regressor Complexity Performance')
    pl.plot(max_depth, test_err, lw=2, label = 'Testing Error')
    pl.plot(max_depth, train_err, lw=2, label = 'Training Error')
    pl.legend()
    pl.xlabel('Maximum Depth')
    pl.ylabel('Total Error')
    pl.show()
```

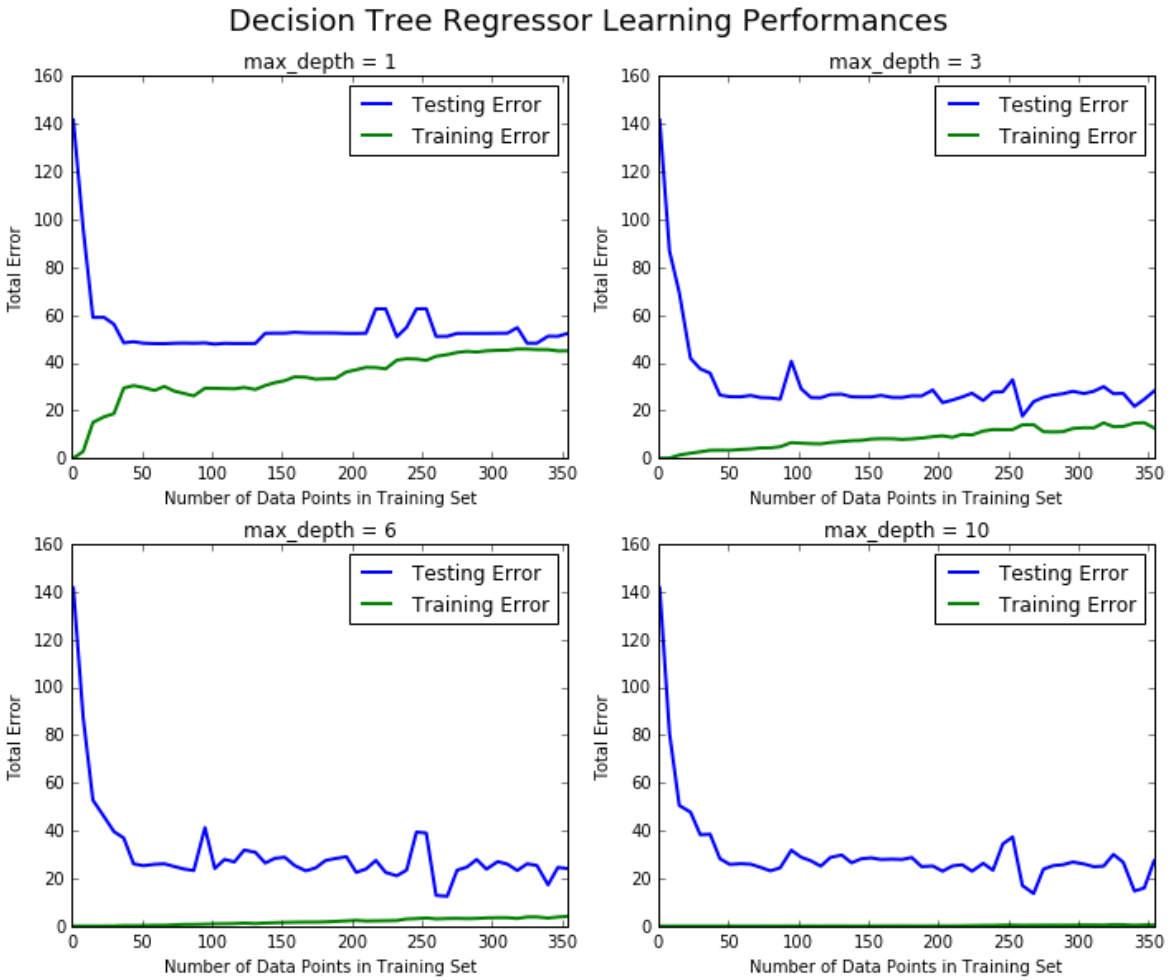
Analyzing Model Performance

In this third section of the project, you'll take a look at several models' learning and testing error rates on various subsets of training data. Additionally, you'll investigate one particular algorithm with an increasing `max_depth` parameter on the full training set to observe how model complexity affects learning and testing errors. Graphing your model's performance based on varying criteria can be beneficial in the analysis process, such as visualizing behavior that may not have been apparent from the results alone.

In [62]:
```
learning_curves(X_train, y_train, X_test, y_test)
```

```
Creating learning curve graphs for max_depths of 1, 3, 6, and 10. . .

C:\WinPython-32bit-2.7.10.3\python-2.7.10\lib\site-packages\ipykernel\__main__.py:24: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
C:\WinPython-32bit-2.7.10.3\python-2.7.10\lib\site-packages\ipykernel\__main__.py:27: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
```



Question 7

Choose one of the learning curve graphs that are created above. What is the max depth for the chosen model? As the size of the training set increases, what happens to the training error? What happens to the testing error?

Answer: I am choosing the bottom left graph. The max depth is 6. As the size of the training set increases, the training error increases, although at a relatively small rate. As the size of the training set increases, the testing error initially decreases rapidly, and then basically pleateaus, with some variance in the plateau as more data continues to be added.

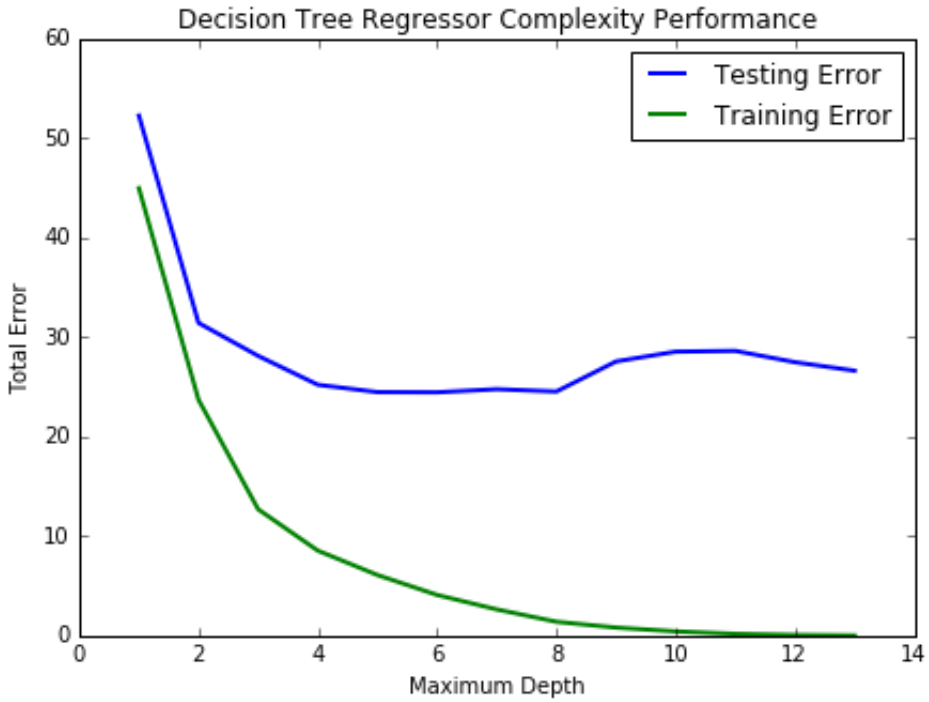
Question 8

Look at the learning curve graphs for the model with a max depth of 1 and a max depth of 10. When the model is using the full training set, does it suffer from high bias or high variance when the max depth is 1? What about when the max depth is 10?

Answer: When the max depth is 1, the model suffers from high bias. This is because the model is underfitted. It is analagous to using a small number of features, where when one uses a small number of features one is making a simplistic model that does not account for all the patterns in the training data. Although tree depth does not necessarily equal number of features analyzed, if we look at decision tree regression as a game of 20 questions we are only asking 1 question, and therefore are not narrowing in on the nature of the training data. Therefore training error is high and the model is biased, and will also give us a large testing error. When the max depth is 10, the model suffers from high variance. It is overfitted to the training data, when the testing data may not be applicable to some of the complex patterns found in the training data. Therefore training error is low, but testing error is higher than the optimal level.

In [63]:
```
model_complexity(X_train, y_train, X_test, y_test)
```

```
Creating a model complexity graph. . .
```



Question 9

From the model complexity graph above, describe the training and testing errors as the max depth increases. Based on your interpretation of the graph, which max depth results in a model that best generalizes the dataset? Why?

Answer: I believe that the model that best generalizes the dataset is at max depth = 6. This is because both training and testing error are obviously too high at max depth 1 and 3. So the choice is between 6 and 10. At 6 the training error is around 4 or 5, and the testing error is around 25. If you change the max depth to 10, the training error decreases a few points to around 1, but the testing error increases to around 30. Therefore the overall lowest error is at max depth 6.

Model Prediction

In this final section of the project, you will make a prediction on the client's feature set using an optimized model from `fit_model`. *To answer the following questions, it is recommended that you run the code blocks several times and use the median or mean value of the results.*

Question 10

Using grid search on the entire dataset, what is the optimal `max_depth` parameter for your model? How does this result compare to your intial intuition?
Hint: Run the code block below to see the max depth produced by your optimized model.

In [64]:
```
print "Final model optimal parameters:", reg.best_params_
```

```
Final model optimal parameters: {'max_depth': 6}
```

Answer: The optimal parameter is at max depth 6, according to the grid search on the entire dataset. This is in line with my intuition.

Question 11

With your parameter-tuned model, what is the best selling price for your client's home? How does this selling price compare to the basic statistics you calculated on the dataset?

Hint: Run the code block below to have your parameter-tuned model make a prediction on the client's home.

In [65]:
```
sale_price = reg.predict(CLIENT_FEATURES)
print "Predicted value of client's home: {0:.3f}".format(sale_price[0])
```

```
Predicted value of client's home: 20.766
```

Answer: The best selling price for the client's home is 20.766 (in thousands.) This is below the average price for a home in Boston, both when using the median, and when using the mean.

Question 12 (Final Question):

In a few sentences, discuss whether you would use this model or not to predict the selling price of future clients' homes in the Greater Boston area.

In [ ]:
```

```

**Answer: This model does well for a basic understanding of regressive prediction, however I believe there are improvements that can be made to the model. For example, one of the simpler ways to improve the model would be to add more data. Adding more data would limit the effect of outliers as well as provide more training data in order to create a better fit. Max depth might change but that is the point! You could add more data based on the same 13 features, or you could add more features as well. Another way to supplement the data would be to arrange the data by year, so that trends over time could be analyzed. At that point it might be useful to look at other algorithms. The decision tree regression algorithm that was used has specific disadvantages that might come into play. For example, decision tree algorithms can create overly complex models. In other words they are prone to overfitting. They look at the optimal conditions at local nodes (greedy) and therefore may not make the optimal tree globally. Scikit's documentation lists mitigation factors such as training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement (http://scikit-learn.org/stable/modules/tree.html#tree (http://scikit-learn.org/stable/modules/tree.html#tree)). Evolutionary algorithms are one way to avoid a focus on locally node decisions. There are also other algorithms that may be useful such as decision graphs, or bayesian cart model search.
(https://en.wikipedia.org/wiki/Decision_tree_learning#Alternative_search_methods (https://en.wikipedia.org/wiki/Decision_tree_learning#Alternative_search_methods))