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 [1]:

```
# Importing a few necessary libraries
import numpy as np

import matplotlib.pyplot as pl
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.metrics import mean_squared_error
from sklearn.metrics import make_scorer
from sklearn.grid_search import GridSearchCV
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 pr
CLIENT_FEATURES = [[11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385, 24,

# 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 [2]:

```
feature = np.asarray(housing_features)
prices = np.asarray(housing_prices)

# Number of houses in the dataset
total_houses = feature.shape[0]

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

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

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

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

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

# Standard deviation of housing values of the dataset
std_dev = np.std(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 I chose are RM, TAX and ZN.

RM refers to the average number of rooms per dwelling. A great number of rooms implies higher house value.

TAX refers to full-value property-tax rate per $10,000. A higher tax rate implies higher house value.

ZN refers to the proportion of residential land zoned for lots over 25,000 sq.ft.. The lower proportion of residential land the higher the house value as it may imply the house is near central business district.

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 [3]:

```
print CLIENT_FEATURES
```

```
[[11.95, 0.0, 18.1, 0, 0.659, 5.609, 90.0, 1.385, 24, 680.0, 20.
2, 332.09, 12.13]]
```

Answer:

RM: 5.609
TAX: 680.0
ZN: 0.0

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 the `shuffle_split_data` function is working if the statement "*Successfully shuffled and split the data!*" is printed.

In [4]:

```
# Put any import statements you need for this code block here

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

    # Shuffle and split the data
    X, y = shuffle(X, y, random_state=0)

    X_train = X[:len(X)*7/10]
    y_train = y[:len(X)*7/10]
    X_test = X[len(X)*7/10:]
    y_test = y[len(X)*7/10:]

    # 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_features, ho
    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 3

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

Answer:

We want to develop and evaluate a predictive model using the given dataset. By spliting the data into training and testing subsets, we are able to train the predictive model using the training set with feature data and labels, and subsequently varify the predictive model by comparing the predicted values from the test feature data and labels.

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 the `performance_metric` function is working if the statement "*Successfully performed a metric calculation!*" is printed.

In [5]:

```
# Put any import statements you need for this code block here

def performance_metric(y_true, y_predict):
    """ Calculates and returns the total error between true and predicted valu
        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 main advantage of the mean-square error is the analytical tractability that comes with it. Many problems have simple solutions, either as closed-form formulas or semi-closed-form algorithms. The mathematical ease is related to the fact that problems phrased in terms of minimizing the mean-square error are equivalent to calculating projections in linear functional spaces.

While predicting housing price is a regression model, Accuracy, Precision, Recall and F1 score are only applicable for classification problem and hence not application for this case.

Besides,

Accuracy is the number of true positive and true negative devided by the total population. Accuracy is not ideal in this case as the data size is small and may be skewed.

Precision is the number of correct positive results divided by the number of all positive results where Recall is the number of correct positive results divided by the number of positive results that should have been returned. Both Precision and Recall only take in consideration of positive cases and not negative cases. Hence it is not the most ideal for analyzing the total error.

The F1 score is a weighted average of the precision and recall, where F1 = 2 (precision recall) / (precision + recall). For predicting housing prices with continuous regression model, we care more about how close the prediction is than having a good measure of how accurate or consistent the model can make a close prediction.

An alternative measure of Mean Absolute Error which may have an advantage over MSE for being less affected by outliers. However, I choose MSE over MAE becuase MAE is not differentiable which it may make minimization harder.

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 the `fit_model` function is working if the statement "*Successfully fit a model to the data!*" is printed.

In [6]:

```
# Put any import statements you need for this code block

def fit_model(X, y):
    """ Tunes a decision tree regressor model using GridSearchCV on the input
        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(mean_squared_error, greater_is_better=False

    # Make the GridSearchCV object
    reg = GridSearchCV(estimator=regressor, param_grid=parameters, scoring=sco

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

    # Return the optimal model
    return reg.best_estimator_


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

```
Successfully fit a model!
```

Question 5

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

Answer:

Grid search algorithm is simply an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm. It is used when parameters that are not directly learnt within estimators. they can be set by searching a parameter space for the best score exhaustively.

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-vaidation is used to evaluate a trained estimator's performance, in order to prevent overfitting. It holds out part of the available data as a test set X_test, y_test. for Cross validation, especially on K-fold CV - the data set is divided into k subsets, and the holdout method is repeated k times. Each time, one of the k subsets is used as the test set and the other k-1 subsets are put together to form a training set. Then the average error across all k trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set k-1 times. Cross validation is useful because it maximize both the training and testing data so that the data we can use to provide best learning result and best validation - this is extremely useful when the dataset is limited in size. K-fold cross validation makes full use of the dataset and helps grid search validate each combination of chosen hyperpamapeters and compare their score to conclude a best set of hyperparameters by running grid search for k times and select the most frequent one.

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 [7]:

```
def learning_curves(X_train, y_train, X_test, y_test):
    """ Calculates the performance of several models with varying sizes of tra
        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.rint(np.linspace(1, len(X_train), 50)).astype(int)
    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 ma
            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

            # 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,
    fig.tight_layout()
    fig.show()
```

In [8]:

```
def model_complexity(X_train, y_train, X_test, y_test):
    """ Calculates the performance of the model as model complexity increases.
        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
        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_train))

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

    # 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

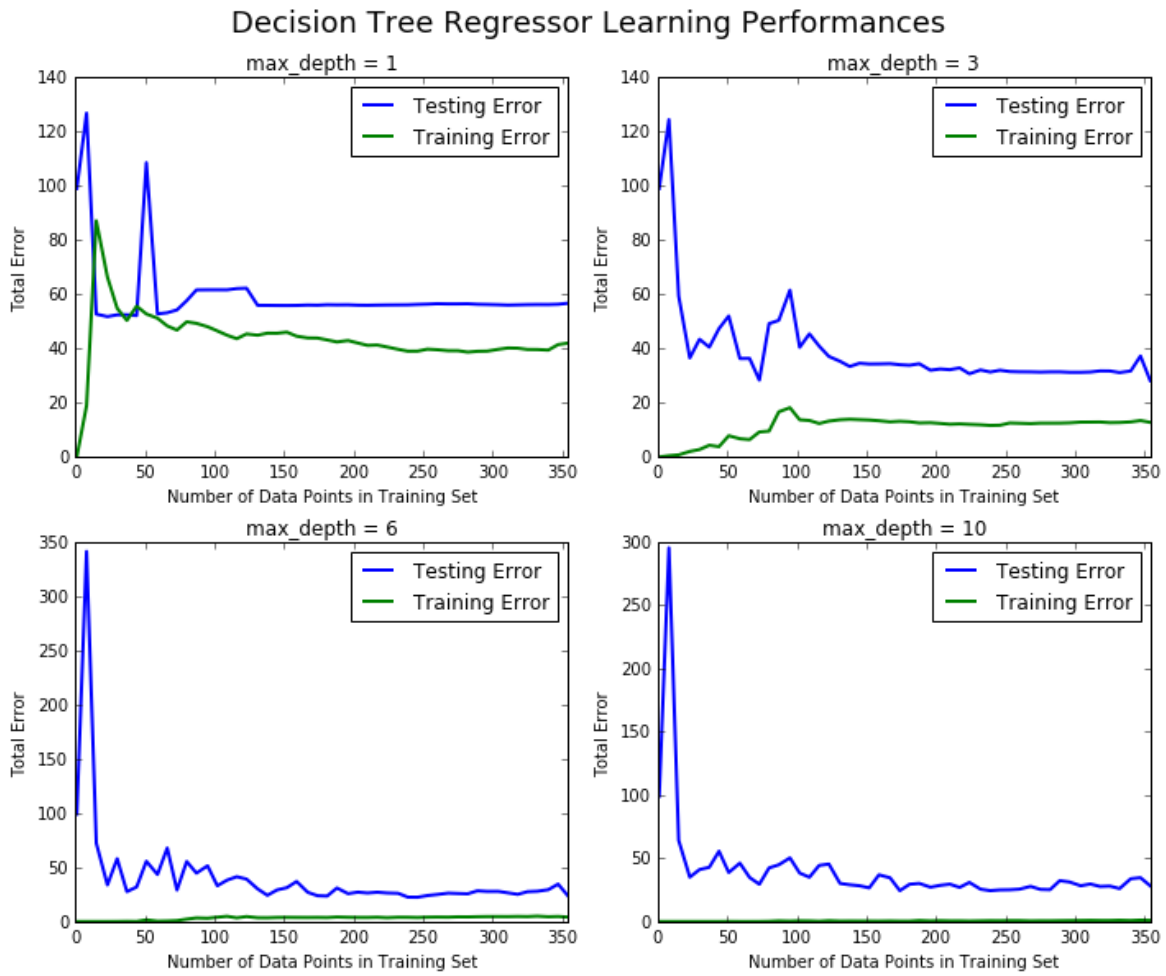
In this third section of the project, you'll take a look at several models' learning and testing error rates on various subsets of training data. Additionally, you'll investigate one particular algorithm with an increasing `max_depth` parameter on the full training set to observe how model complexity 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 [9]:

```
learning_curves(X_train, y_train, X_test, y_test)
```

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

/Users/junhua_liu/Playground/udacity/MLND/venv/p1/lib/python2.7/s
ite-packages/matplotlib/figure.py:397: UserWarning: matplotlib is
currently using a non-GUI backend, so cannot show the figure
  "matplotlib is currently using a non-GUI backend, "
```



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 choose the oen with Max_depth = 6. Though graphically 6 gives a greater gap but it uses a diferent grip. It seems to give the best result among all.

Based on the observation of the plots above, as the size of the training set increases, the model is underfitted as testing error is unreasonably high. At about 10 data points, the testing error decreases exponentially until the number of data points reach 20, and the total errors fluctuates between 30 to 70.

After 175 data points, the testing error remains stable at a total error of about 300.

To give more context, when the training set is small, the trained model can essentially “memorize” all of the training data. As the training set gets larger, the model won’t be able to fit all of the training data exactly.

The opposite is happening with the test set. When the training set is small, then it’s more likely the model hasn’t seen similar data before. As the training set gets larger, it becomes more likely that the model has seen similar data before.

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?

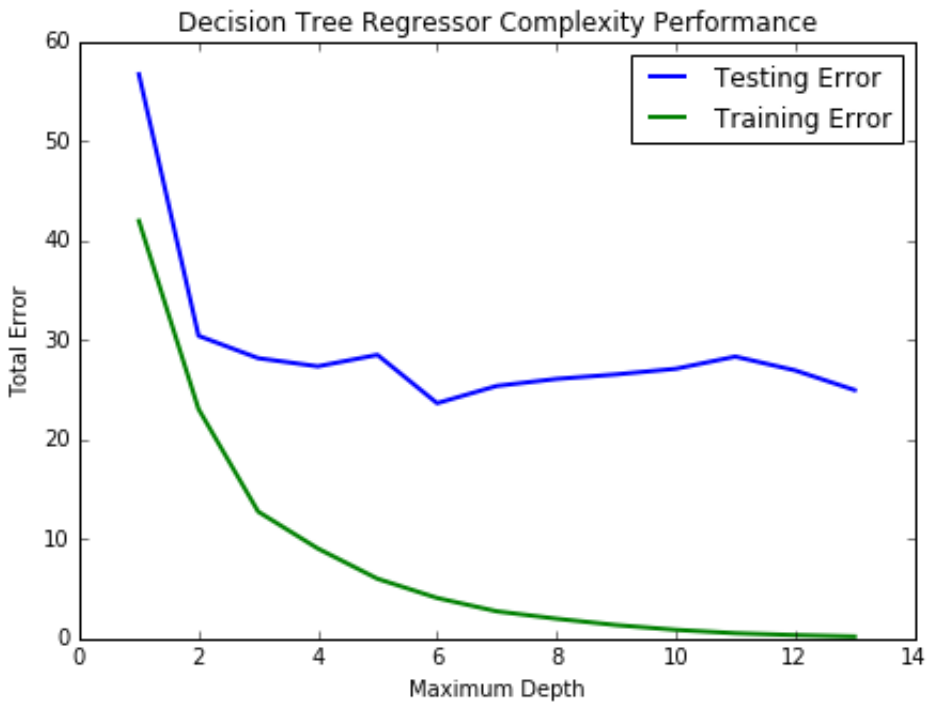
Answer:

When the max depth is 1, it suffers from high bias as both the training and testing errors converge at a high value (~50). When the max depth is 10, both training and testing errors dont converge and remain an observable gap. It appears to suffer a high variance.

In [10]:

```
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:

As the max depth increases, the both error curves drop quickly until max depth reaches 2. After that, the testing error drops further but gentally until 6 and bounce back up. On the other hand, the training error continues to drop quickly with a reducing gradient until the maximum depth reaches 12 and the total training error approaches 0. Baesd on my interpretation of the graph, a max depth of 6 results in a model that best generaizes the dataset. At a maximum depth of 6, the estimate is expected to perform drastically better than any smaller value, and similarly with any larger value as the testing error goes flat. It gives the lowest complexity of the training for a relatively optimal estimator.

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`. When applying grid search along with cross-validation to optimize your model, it would typically be performed and validated on a training set and subsequently evaluated on a **dedicated test set**. In this project, the optimization below is performed on the *entire dataset* (as opposed to the training set you made above) due to the many outliers in the data. Using the entire dataset for training provides for a less volatile prediction at the expense of not testing your model's performance.

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 [11]:

```
print "Final model has an optimal max_depth parameter of", reg.get_params()['m
```

```
Final model has an optimal max_depth parameter of 6
```

Answer:

The optimal max_depth is 6. This result is the same as my initial 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 [12]:

```
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 is 20.766. The predicted selling price is 10% lower than the mean house price of 22.533.

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.

Answer:

Yes I would use this model to predict the selling price of future clients' homes in the Greater Boston as it takes in more parameters in the process of estimating the selling price as compared to simply finding the average. However, I will also use this model as a baseline and attempt to try a few other regression algorithms and gathering more data to improve the performance of the estimator.