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 [3]:
```
# 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 no
tebook)
%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]]

# 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 [4]:
```
# Number of houses in the dataset
total_houses = housing_prices.size

# Number of features in the dataset
total_features = housing_features[0].size

# 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:

Not having done the analysis up front, I can't speak with certainty about which features are going to be significant, but I can give some guesses and why I think they're reasonable.

DIS *(weighted distances to five Boston employment centres)* This is one of the things I care most about when moving, because a long commute is a big negative impact on my life. Given the way I've watched housing prices shift in areas I've lived in before as medium-to-high salary employers move into an area, people value living fairly close to work, and this is reflected in what they're willing to pay for a home. San Francisco's housing market is probably the most extreme example of this right now.

RM *(average number of rooms per dwelling)* This seems an easy argument to make. More rooms per house, except in rare cases of extreme internal compartmentalization, means bigger houses. Bigger houses each take up proportionally more of the actual underlying scarce resource (the land), require more raw materials to build and maintain, and are more likely to be comparatively up scale rathern than more utilitarian-style houses because if you're trying to really save money you are not likely to be looking for a large house.

AGE *(proportion of owner-occupied units built prior to 1940)* Old houses require more upkeep and in some cases major applicance/structural repairs and replacements. There are also more likely to be the occasional house in a neighborhood that has fallen into total disrepair in neighborhoods full of old houses (it's hard for a house built a year or two ago to already be decrepit and abandoned). This should drive down prices in a neighborhood, unless balanced by a significant demand for houses in that location due to other concerns regardless of the state of the building, or the area is upscale enough that all property owners have consistenly been renovating.

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 [5]: 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:

DIS *(weighted distances to five Boston employment centres)* 1.385

RM *(average number of rooms per dwelling)* 5.609

AGE *(proportion of owner-occupied units built prior to 1940)* 90.0%

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 [23]:
```
# Put any import statements you need for this code block here
from sklearn.cross_validation import train_test_split

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_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)

    # 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, 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 3

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

Answer:

If we trained on all the data we have available, then it would be difficult to ascertain whether the model was generalizing well. Certainly we could give it some data points from the training set to see how well it categorized them, but we absolutely expect it to do quite well with thoses inputs because it used them to decide how to model the algorithm; the model would basically have "memorized" the answer. In order to make sure our model isn't overfitting to the data it has available, we need to test how well it generalizes by giving it data it hasn't seen yet (and, importantly, data that we know what the correct prediction is so that we can measure the error). The standard way to do this is to reserve some of the available data set from the training run.

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 [24]:
```
# Put any import statements you need for this code block here
from sklearn.metrics import mean_squared_error

def performance_metric(y_true, y_predict):
    """ Calculates and returns the total error between true and pre
dicted 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 calculatio
n."
```

```
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: I intitially filtered out Accuracy, Precision, Recall, and F1 score because they all seem better suited to discrete classifiers than regression algorithms. How do I decide whether a value was "accurate"? I can't say that it was or was not, only how close it was to the correct price. MSE and MAE both seem reasonable. I actually considered Mean Absolute Error first because it should be less impacted by the occasional *way* off price (since it's not squaring the error calculation), but decided that MSE is the standard error measure for regressions, and that if I wasn't really sure that picking another one made sense, that seemed the most responsible choice (and easiest to understand and compare to other studies for other analysts reviewing the project, since it would be the error measure they would be most accustomed to).

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 [25]:
```
# Put any import statements you need for this code block
from sklearn.metrics import make_scorer, mean_squared_error
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(score_func=mean_squared_error, greater_is_better=False)

    # Make the GridSearchCV object
    reg = GridSearchCV(regressor, parameters)

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

    # Return the optimal model
    return reg.best_estimator_


# 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."
```

```
Successfully fit a model!
```

Question 5

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

Answer: The grid search algorithm gives us a way to tune learner parameters without playing guess and check via manual parameter optimization. Given a set of options for each of the parameters you want to tune, it will generate an exhaustive grid of the permutations and fit/score a learner with each one. In this manner you can find out which combination of the provided parameter options gets you a learner with the best fit.

With regard to when it is applicable, I think that the answer this project *wants* is something like "when you don't have a good idea of what reasonable parameter values are for your model" because it will be more exhaustive than manual optimization. Based on some additional extracurricular reading though (http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf (http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf)) I think it may be more accurate to say that grid search is applicable in low dimensional space, and that as your number of features grows and the potential for more local min/max issues arises, a randomized search might actually be preferable (I don't have enough experience to take a strong strand here, but the reasoning seems intuitively correct to me).

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 the process of taking a data set and splitting it into many subsets (rather than one large training set and one small testing set) and iteratively using the inverse of each subset as training data and then the subset itself as testing data. In this way a researcher can take the average of the error score for each subset to get a general idea of fit quality for a model that is resiliant to a biased training/test split. This is valuable when doing parameter tuning (via grid search) because if you used a single static training/test split than some of the difference in scoring for a given set of parameters for a model might be because of the quirks of the way the dataset was split. Using the average of multiple passes with different training/test splits for the data means that you don't end up evaluating a particular parameter permutation as superior simply because it happened to fit really well for one particular training/test division.

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 [9]:
```
def learning_curves(X_train, y_train, X_test, y_test):
    """ Calculates the performance of several models with varying s
izes of training data.
        The learning and testing error rates for each model are the
n 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 differe
nt 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 t
ree 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], regresso
r.predict(X_train[:s]))

            # Find the performance on the testing set
            test_err[i] = performance_metric(y_test, regressor.pred
ict(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', f
ontsize=18, y=1.03)
    fig.tight_layout()
    fig.show()
```

In [10]:
```
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 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_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()
```

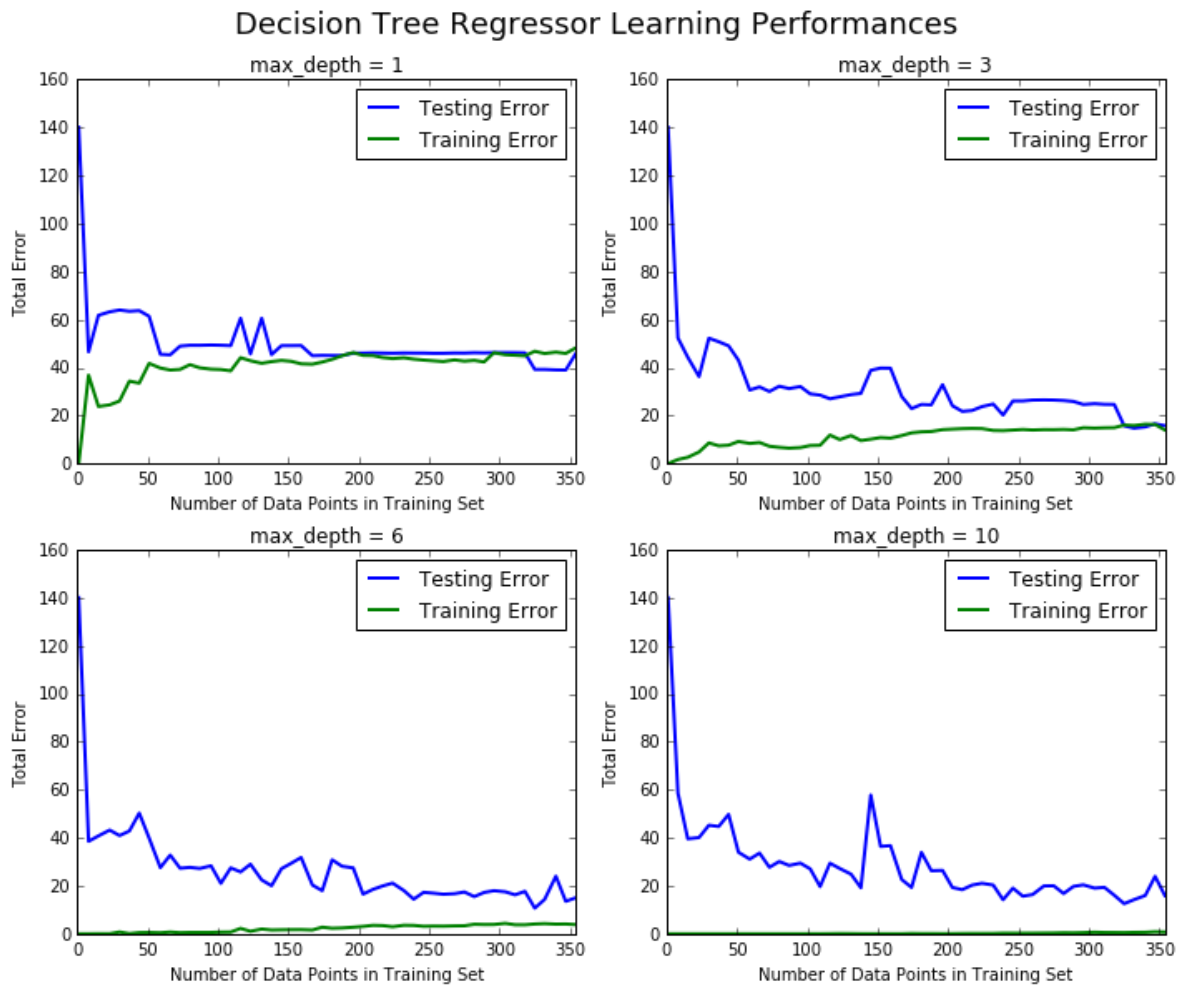
Analyzing Model Performance

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 [11]:
```
learning_curves(X_train, y_train, X_test, y_test)
```

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

/usr/local/lib/python2.7/site-packages/matplotlib/figure.py:397: U
serWarning: matplotlib is currently using a non-GUI backend, so ca
nnot 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'm looking at the learning curve graph with a max depth of 10 (lower right graph). As the number of data points goes up, the training error reamins very low, but increases very slightly (probably still around or less than 1). Even with the small increase, it still appears to me intuitively that the algorithm is likely overfit at 350 data points in the training set. The Testing Error begins quite high with little data, and trends downwards as the size of the training set increases. It doesn't appear that it's going to get much better than 16-20, no matter how much additional data we add.

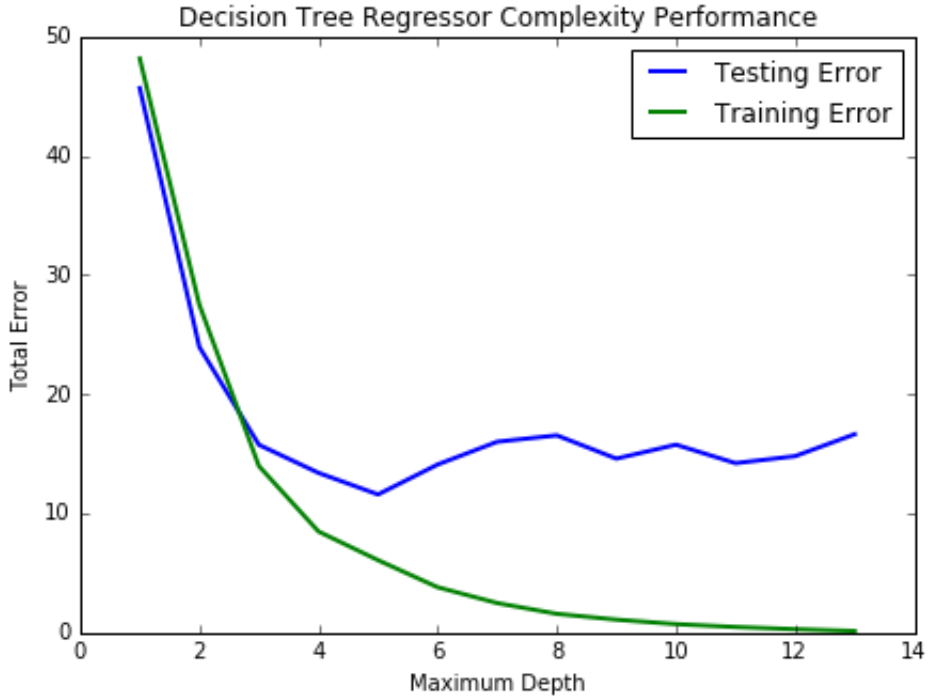
Question 8

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 set to 1, the model seems to suffer from high bias. You can continue to feed it more and more data, and it still has high error rates even in the training set. When the max depth is set to 10, it seems to suffer from high variance; adding data is still making the test error rate rise and fall non-trivially while increasing the data points in the training set from 300 to 350, but the gap between training and testing errors is still quite high, especially because the errors for the training set stays at almost 0. It may currently be overfit to the training data, even at 350.

In [12]:
```
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've not interpreted one of these complexitey graphs before, but I'm going to go with 5, and my reasoning is as follows: the ultimate goal is to make the learning algorithm as accurate as possible for *unseen* data, which means the testing error is more important to examine than the training error. Increasing the depty from 1 to 2 drops the error rate for the testing set significantly, as does 2-3, 3-4, and 4-5. After this, increasing the max depth only brings down the error rates for the training set, but the error rates for the testing set kind of plateau. That additional depth isn't buying us any better prediction, it's just taking up additional time/complexity in training and probably resulting in an overfit algorithm.

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 [26]:
```
print "Final model has an optimal max_depth parameter of", reg.get_params()['max_depth']
```

```
Final model has an optimal max_depth parameter of 4
```

Answer: I ran the model 14 times, coming out with the following optimized max_depth values:

8,4,6,6,5,9,4,4,4,6,10,9,5,4 mean: 6.0 median: 5.5

The model appears to have an optimal max depth of a little less than 6, which is only slightly higher than my prediction of 5 based on the complexity graph, and based on that I still think 5 is a valid answer for best depth.

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 [33]:
```
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: 20.776 is the predicted value of the client's home. This is slightly below the mean and median of the data set, but still well within one standard deviation of 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.

Answer: If it were still 1993, I would consider it (1993 is the year the data was collected I think, or at least the year it was donated to make the open dataset available). If we're talking about 2016, absolutely not, the data is 23 years out of date at least. Even if the underlying factors for relative value remained mostly stable, inflation and real estate absolute values alone have changed enough to make the predictions worthless. Even if it *were* 1993, I'd take these predictions with a grain of salt; the best error rates for the test sets were just below 20%. That's definitey better than guessing, but not good enough that I'd stake a large financial decision on it's accuracy. I'd maybe use it as an anchoring point to start a discussion around what a good price would be.

In [ ]:
```
```