boston_housing

March 31, 2016

1 Machine Learning Engineer Nanodegree

1.1 Model Evaluation & Validation

1.2 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, which is provided by the **UCI Machine Learning Repository**.

2 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, 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 [2]: # 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

# 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!
```

3 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.

3.1 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 [3]: # Number of houses in the dataset
        total_houses = housing_prices.size

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

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

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

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

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

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

        # 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
```

3.2 Question 1

As a reminder, you can view a description of the Boston Housing dataset here, 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: I chose the features CRIM = crime rate per capita, PTRATIO = pupil-teacher ratio, and TAX = property tax rate.

I get the impression that high crime areas are typically poorer, and this would likely be reflected in the property price. CIM indicates the per-capita crime rate per town. So large values of CRIM might indicate cheaper housing. On the other end of the spectrum, high pupil-teacher ratio's migh indicate expensive housing. School districts often dictate where new parents want to live, and new parents buy houses. "Good" school districts often boast higher PT ratios. Therefore, high PT Ratio's will increase the demand for housing from parents. I would expect this to be reflected in the housing price. Lastly, TAX indicates the full value property tax-rate. My guess is that taxes are lower in cheaper neighborhoods, and higher in richer areas. I don't know if this is true, however I could imagine poor people being much less comfortable with taxes than rich people. As a result, rich neighborhoods might have higher tax rates.

3.3 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 [46]: 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: 11.95 for CRIM, 680.0 for TAX , and 20.2 for PTRATIO

4 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.

4.1 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
        from sklearn import cross_validation

        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 = cross_validation.train_test_split(X,y,test_size=0.3)

            # 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!
```

4.2 Question 3

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

Answer: We must concoct some test the ensure that we are not overfitting (i.e. mitigate variability). By definition, overfitting occurs when the model does not generalize well to data outside of the training set. Therefore we must have some test set which is not involved in training the model so that we can see how well, or not well our model generalizes. This neccesitates that we split the data into a training set and a test set.

4.3 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 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
        from sklearn.metrics import mean_squared_error
        from sklearn.metrics import mean_absolute_error

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

            error = mean_absolute_error(y_true, y_predict)
            #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."
```

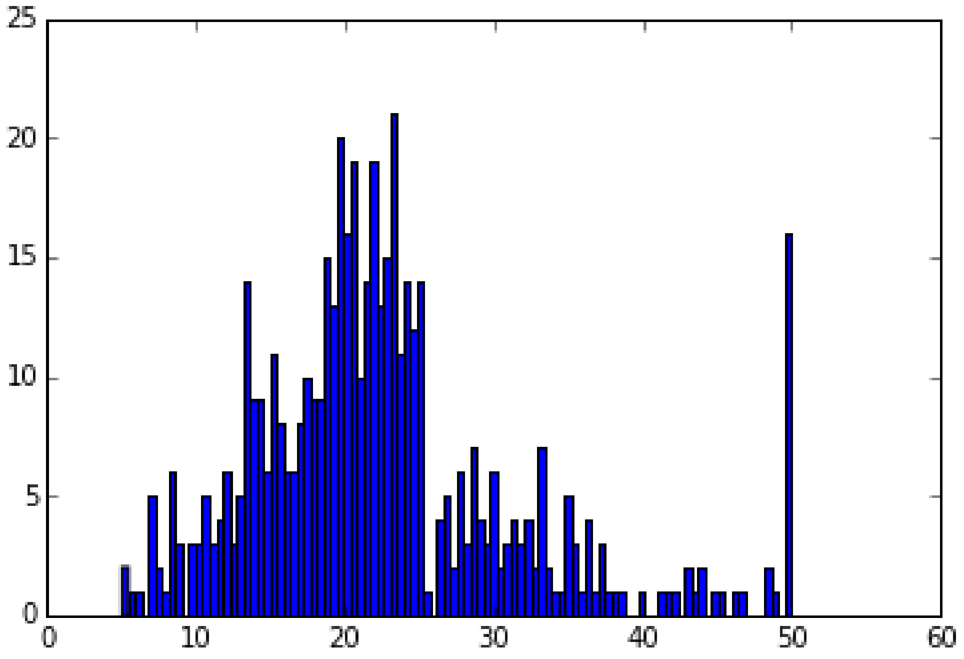
```
Successfully performed a metric calculation!
```

4.4 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: As this is a regression problem, the only two appropriate candidates for an error metric are the mean squared error (MSE) and the mean absolute error (MAE). For me, the MSE is preferable by default, as the gradient calculations are generally more simple. However, if there are outliers or fat tails in the data, then MSE is very sensitive because it squares large numbers to make super large numbers. For this reason, the MAE might be prefereable in the presence of outliers and fat tails. In this case the data seems to have a large spike on the end (see histogram below)

```
In [6]: n, bins, patch = pl.hist( housing_prices, 100)
```



Answer (continued): I suspect this spike at the end is due to the fact that houses over some threshold were just lumped into the same category (i.e the final bin in the histogram represents houses of price "x" and higher, as opposed to houses in some finite range). I chose MAE as my error metric because I did not want this large spike at the end of the histogram to bias the results too much.

4.5 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. - Build a GridSearchCV object using `regressor`, `parameters`, and `scoring_function`. See the sklearn documentation on GridSearchCV.

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 [7]: # Put any import statements you need for this code block
        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(mean_absolute_error, greater_is_better=False)

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

            # 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!
```

4.6 Question 5

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

Answer: The grid search algorithm as a method to find an optimal choice of parameters for a model. In the example above the decision tree class has a parameter "max depth" which must be provided to each instance. The grid search algorithm tests a list of values for "max depth", and finds the "best" one, where "best" is determined by a scoring function. In this case the scoring function is given by the mean absolute error achieved under that choice of parameters. Grid search is applicable whenever there are free parameters in the model and it is unclear as to how best to choose them.

4.7 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: When using grid search, minimizing the some scoring function (such as MSE or MAE) by allowing a parameter to vary (such as "max depth") might produce models which are guilty of over fitting. This is because over fit models often produce smaller errors with respect to a training set. Cross validation is the process of splitting your data into a test set and a training sets so that one can test the model on a test set and ensure and check for over-fitting. Therefore cross-validation can be crucial after doing grid-search.

5 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 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.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 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 [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()
```

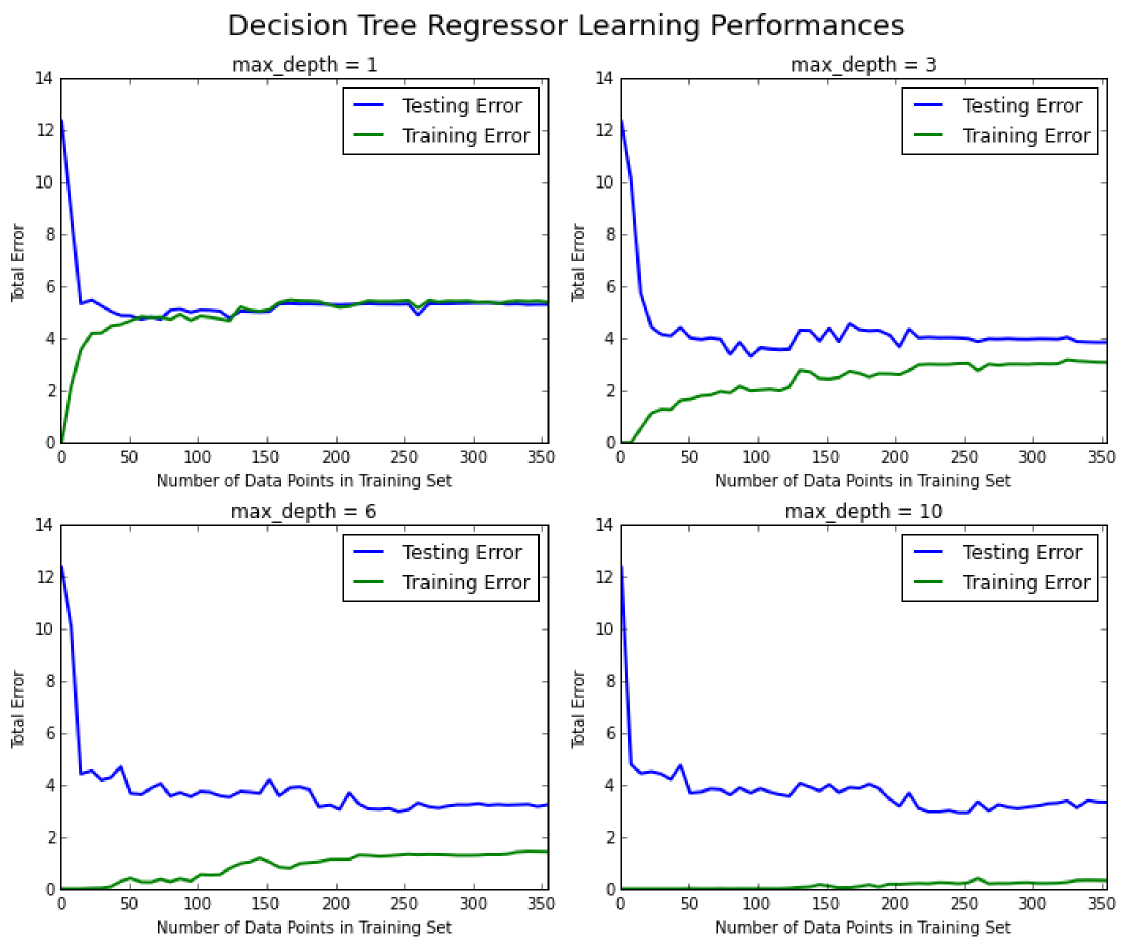
6 Analyzing Model Performance

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:387: UserWarning: matplotlib is currently us
  "matplotlib is currently using a non-GUI backend, "
```



6.1 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: For the top right graph, the max depth is 3. As the size of the training set increases the training error seems to increase as well. Perhaps this is because the model is being asked to fit a larger amount of data without any increase in complexity. In contrast, the testing error appears to decrease initially, and then platueau. Assuming the data independent and identically distributed with respect to some fixed distribution, we should not expect testing error to decrease substantially when the training set has sufficiently sampled this distribution.

6.2 Question 8

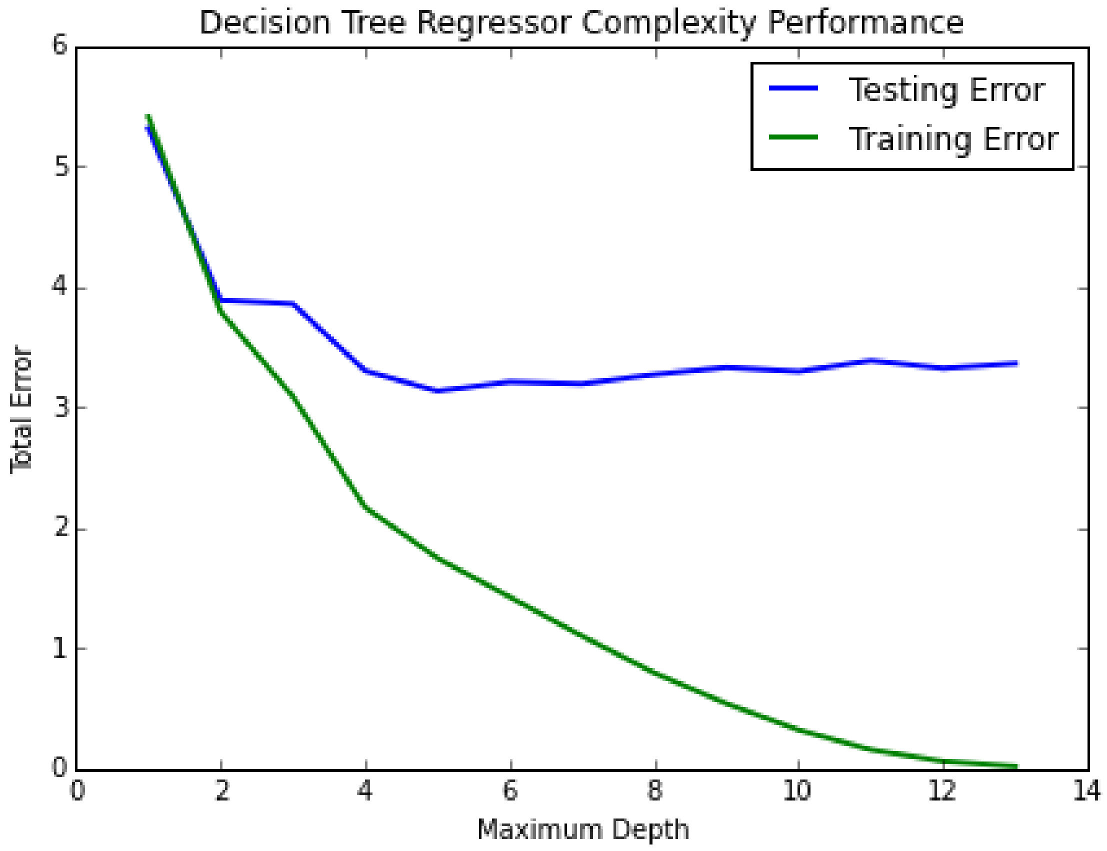
Look at the learning curve graphs for the model with a max depth of 1 and a max depth of 10. When the model is using the full training set, does it suffer from high bias or high variance when the max depth is 1? What about when the max depth is 10?

Answer: When the max depth is 1 one might suspect that the model could suffer from high bias. If it does, we should be able to see high test and training errors for large training data sets. In this case we see that training error for a max depth of 6 is less than half that at a max depth of 1 when the training set size is 350. The testing error is also about 30% lower. Therefore, it seems that there is bias.

In contrast, with a max depth of 10 the model appears to suffer from high variance. The testing errors are much larger than the testing errors, even when using 350 data points to train. It appears the model does not generalize well when the max-depth is 10.

```
In [12]: model_complexity(X_train, y_train, X_test, y_test)
```

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



6.3 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: We should generally expect the training error to be less than the testing error. As the maximum depth increases, the model is able to better fit the data and so training error drops. However, testing error does not drop after a maximum depth of 6 because overfitting begins to dominate the testing error. It seems that somewhere between 4 and 7 is a good maximum depth for this model because larger max depths fail to decrease testing error (the ultimate error we wish to minimize) while lower maximum depths suffer high bias (i.e. both types of errors are relatively high).

7 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.

7.1 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 [13]: print "Final model has an optimal max_depth parameter of", reg.get_params()['max_depth']
```

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

Answer: The optimal max depth appears to be 6. Yay! This is consistent with the reasoning given in question 9.

7.2 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 [14]: 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
```

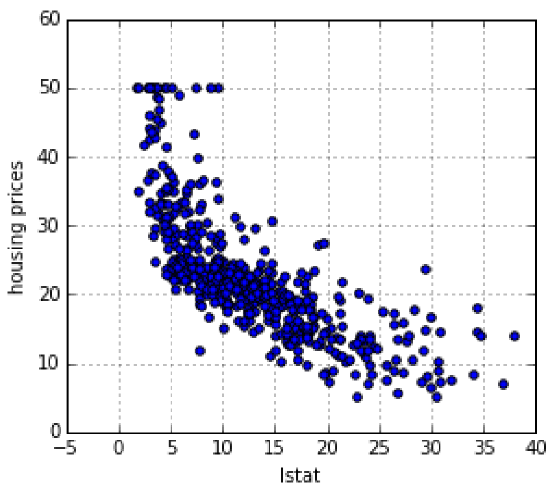
Answer: It appears the best selling price for my client's home is $20,766. This is well within a standard deviation ($9,188) from the mean ($22,533). So this client is very average.

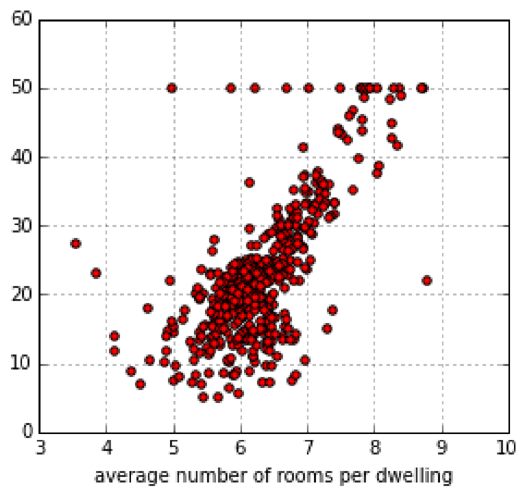
7.3 Question 12 (Final Question):

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: I wouldn't. Given that the typical testing error ($5,000) is roughly a quarter of the average housing price $20,000$, I would be afraid of loosing my clients due to such low accuracy. My guess is that a realtor's human intuition is superior to this model. Moreover, it is possible that linear regression might help things, sice there is a trend in some of the data (see figures)

```
In [22]: rm = housing_features[:,5]
         lstat = housing_features[:,12]
         pl.figure(figsize = (10,4))
         pl.subplot(1,2,1)
         pl.scatter( lstat , housing_prices )
         pl.ylabel('housing prices'),pl.grid(),pl.xlabel('lstat')
         pl.subplot(1,2,2)
         pl.scatter( rm , housing_prices , c='r')
         pl.xlabel('average number of rooms per dwelling'), pl.grid()
         pl.show()
```





Perhaps a linear regression model could be implemented at the leaves of the decision tree. I'm not sure if this would improve things.