# boston_housing

May 16, 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 [1]: # 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
        CLIENT_FEATURES = [[11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385, 2

        # 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!"
```

```
/Users/daiwei/Library/Python/2.7/lib/python/site-packages/matplotlib/font_manager.p
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take
```

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

### 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 choose the DIS, B and LSTAT. The DIS meatures the employment potential of this area. The B meatures the safety to some extent and the LSTAT meatures the average purchasing power and the level of prosperity of the town**

### 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 [14]: 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: 1.385, 332.09, 12.13**

## 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 [61]: # 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 = None
             y_train = None
             X_test = None
             y_test = None

             X_train, X_test, y_train, y_test = cross_validation.train_test_split(X

             # 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
    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: As we need the testing subsets to avoid overfitting or high variance when predicting**

### 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 [62]: # 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 predicted
                 based on a performance metric chosen by the student. """

             #error = [precision_score(y_true,y_predict), recall_score(y_true,y_pre
             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: I think it's the MSE. Because the predict value is continuous, the accuracy, precision, recall and f1 score don't fit this matter. I think MSE is a little better than MAE as it's more 'soft'. As MSE is smaller than MAE when the data has a little bias, like 0.1, the MAE is 0.1 and the MSE is 0.0001. And MSE punish more when the bias is big, like 10, the MAE is 10 and the MSE is 100. So I preferred the MSE.**

### 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 3**. 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 [87]: # 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 in
                 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 =

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

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

        # Return the optimal model
        print reg.best_params_
        print reg.best_score_

        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."
```

```
{'max_depth': 4}
-36.05466437
Successfully fit a model!
```

### 4.6 Question 5

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

**Answer: 'Grid search algorithm do exhaustive search over specified parameter values for an estimator.', when we have chosen a specified model and want to tune the parameters of the model.**

### 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: 'Cross-validation is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set.'It can make more use of the datasets. As it divide the training set into several parts, rotationally let one part as the testing set and the others the training set. CV in using grid search can make the grid search method more reliable, as the fully use of datasets, which is better than just one train_test_split method. I think it's more robustness for coming data.**

## 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 [27]: def learning_curves(X_train, y_train, X_test, y_test):
             """ Calculates the performance of several models with varying sizes o
                 The learning and testing error rates for each model are then plott

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

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

             # We will vary the training set size so that we have 50 different size
             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 wit
                     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.predi

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

                 # 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
             fig.tight_layout()
             fig.show()

In [28]: def model_complexity(X_train, y_train, X_test, y_test):
             """ Calculates the performance of the model as model complexity increa
```

```
        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 de
        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_tra

        # 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 [29]: learning_curves(X_train, y_train, X_test, y_test)

Creating learning curve graphs for max_depths of 1, 3, 6, and 10. . .
0.0
141.596907895
2.82483333333
96.2802122807
```

14.9968888889
58.9663961988
17.344187291
58.9723506229
18.6570505952
56.0895596186
29.4001402027
48.3164332314
30.3992897727
48.7924722451
29.6228197035
48.1840818294
28.3863755136
47.9928423503
30.0583856749
47.9673974119
28.1261035631
48.1560535474
27.1294601716
48.2227185884
26.1006964033
48.1462586472
29.3233471852
48.2892324667
29.2677139745
47.7635647982
29.1728491393
48.1071066
29.0785387931
48.0242442434
29.6228515668
48.0271056759
28.7848174472
48.0650448792
30.4410371771
52.3063249229
31.6549073573
52.3731012323
32.4818150021
52.3614645453
34.0633519154
52.7195436996
33.9285905066
52.4989361937
33.0948731641
52.4299568146
33.2707189516
52.46329767

33.3459370312
52.4151439739
36.1316929902
52.2804233564
37.0300619306
52.2649306715
38.0517128425
52.3559563742
37.9646660552
62.545840603
37.4263543178
62.5394248111
41.0552597464
50.8743064869
41.6590726377
54.8167555483
41.5267086552
62.5354328062
40.9535529055
62.611014749
42.6807432735
50.9935109804
43.3643730917
51.0478611002
44.2894509518
52.2656005688
44.7336553938
52.268490199
44.4685492593
52.2645094288
44.9516432375
52.2772987182
45.1756551542
52.3360726045
45.2381794953
52.3560839192
45.7216570168
54.7043103109
45.7111090396
48.1764385959
45.4922219199
48.177394641
45.4560262702
51.1033072299
44.9208486367
51.0582649134
44.9726872904
52.265274531

0.0
141.596907895
0.04
88.5482236842
1.34922222222
50.3087244152
2.08610766046
41.427185225
2.68514285714
36.1941670023
3.29850600601
33.2908082054
3.36183838384
27.741687167
3.33366666667
26.5386576389
3.65113950456
25.3831040399
3.86513793396
26.6483618984
4.31163549318
25.1368448852
4.34303422619
25.6903879584
4.80010827487
24.0671415334
6.53061342985
32.647678146
6.28470545141
24.1029821189
6.10893933955
26.7400429787
6.00627148914
26.6554038723
6.60468662427
25.3385060003
6.98992113739
25.4711533683
7.33316302162
25.7186486738
7.47691644152
25.6461529765
8.02285564061
25.6456682029
8.17854059148
26.3146956042
8.14856568048
25.4582969753

7.87102426564
25.4146646337
8.07885543435
26.1131449988
8.44307519894
26.0670765917
9.03091672862
28.5743495706
9.37176653137
23.2781840982
8.7466589043
24.3470733448
9.97593416181
27.209095087
9.76273496196
25.66050756
11.3669927376
24.1746088264
12.0078179538
30.5288634211
11.9634975651
32.8673639709
11.9740865379
27.8042132019
13.9826978095
17.6383977483
14.0597048794
23.6948934075
11.0996863899
25.476915563
10.9946641636
26.432519316
11.1191787093
27.0079535043
12.4461490503
27.0712077802
12.7612489392
27.0833492458
12.6954144603
27.0912827701
14.8060820483
29.9850403506
13.2118755828
27.0365966548
13.3131352477
27.1420256374
14.7111755174
17.1861294103

14.8642029292
24.6659851473
12.690436934
28.1091616034
0.0
141.596907895
0.0
91.9232236842
0.0
53.5176973684
0.00760869565217
42.6862828947
0.0217777777778
38.3874342105
0.172306306306
34.8983800439
0.139529220779
28.0169893484
0.204154995331
27.6065642529
0.31200968523
25.3752304123
0.310827922078
28.0324048704
0.503186591947
27.3114509532
0.663683035714
24.0325481859
0.68185892173
23.7130864503
0.782855032318
34.6000535805
0.926451582387
24.1171760138
0.968301315911
27.5890132161
1.03804993905
27.1889687098
1.23046680217
30.3712946398
1.04450268967
26.6015303142
1.27783821915
30.9526204904
1.3974255262
27.5366486613
1.51322203947
26.6399052908

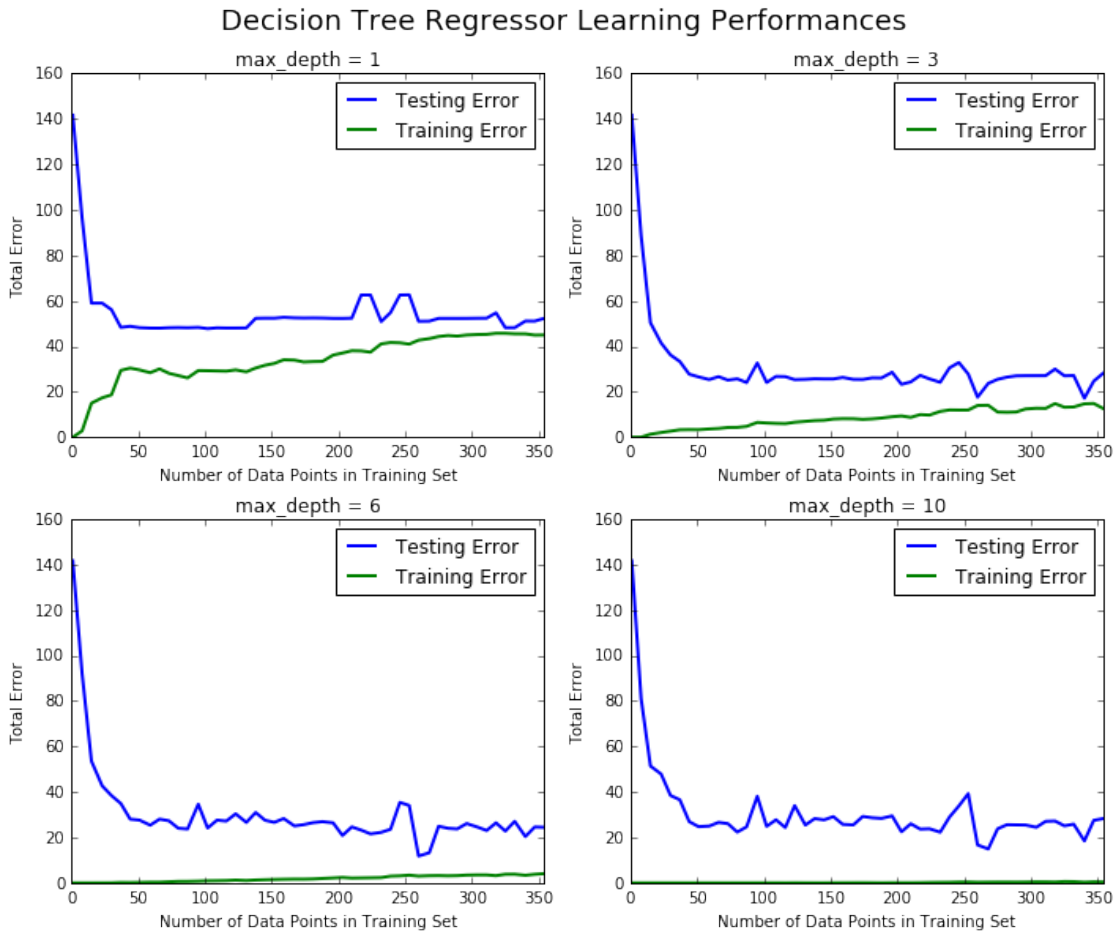
1.59666784099
28.2875282499
1.66391145905
25.0617624142
1.65704461128
25.6207223185
1.74587376282
26.4918198609
1.9546872889
26.9433527917
2.17377372492
26.4096014029
2.39661223165
20.8459200192
2.12794818594
24.6146637435
2.19401350344
23.2112921174
2.25619552772
21.5600961523
2.34297389326
22.2316279991
2.971615117
23.5701260933
3.10206020665
35.4021742841
3.400874179
34.0343116133
2.99127069638
11.8261460019
3.16395316662
13.236036083
3.21158563066
24.9043080107
3.11566451257
24.0189577864
3.19398739046
23.7441227049
3.43212968277
26.1062985454
3.51752972148
24.5727877559
3.50707788765
23.0440060401
3.19100593507
26.4083948453
3.79276858724
22.7793061383

3.82185740098
27.0776639813
3.34192330272
20.3758027441
3.78448157427
24.5870431596
4.07990155031
24.4488510708
0.0
141.596907895
0.0
80.9592105263
0.0
51.3733552632
0.0
47.8407894737
0.0
38.4400657895
0.0
36.5376973684
0.000113636363636
26.9412006579
0.0
24.7419078947
0.0
24.9886842105
0.0
26.6086184211
0.00027397260274
26.0466447368
0.0055
22.3380263158
0.00298850574713
24.6448026316
0.0152105263158
38.0929276316
0.0142156862745
24.8617763158
0.019252948886
27.7528296187
0.00215517241379
24.3406578947
0.0315214866434
33.9845357441
0.0136997455471
25.4299377193
0.00393719806763
28.2808315058

0.00441379310345
27.7172861842
0.0267302631579
29.1423895468
0.0418270440252
25.73805678
0.0550866837753
25.5232556137
0.0687681992337
29.2224155702
0.0507943962115
28.6420555156
0.0216117021277
28.3695858187
0.0584523809524
29.4591081871
0.0495221674877
22.4748901316
0.0525724867725
26.0069224578
0.0896198156682
23.6554612939
0.163300324675
23.8001925507
0.218112631184
22.2899381783
0.250771161893
28.9725480402
0.366452800361
33.8310718303
0.391148820218
39.233137439
0.270479405888
16.6785840111
0.318346686895
14.9307284574
0.347396114505
23.7866732254
0.349274317645
25.5901841644
0.331628373702
25.5177543545
0.364476270914
25.4468999612
0.453548010652
24.4513205077
0.445312225564
27.0725220839

```
0.377758586718
27.2059275818
0.609025811966
25.1075823825
0.557571619813
25.8320439246
0.293397961842
18.4624461813
0.469976546472
27.4840550899
0.417463911716
28.2754034733
```

```
/Users/daiwei/Library/Python/2.7/lib/python/site-packages/matplotlib/figure.py:397:
  "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: I choose the model with max depth 6. As the size of the training set increases, training error becomes a little bigger and become converged at the end and testing error reduces a lot at the begining and become converged at the end.**

### 6.2 Question 8

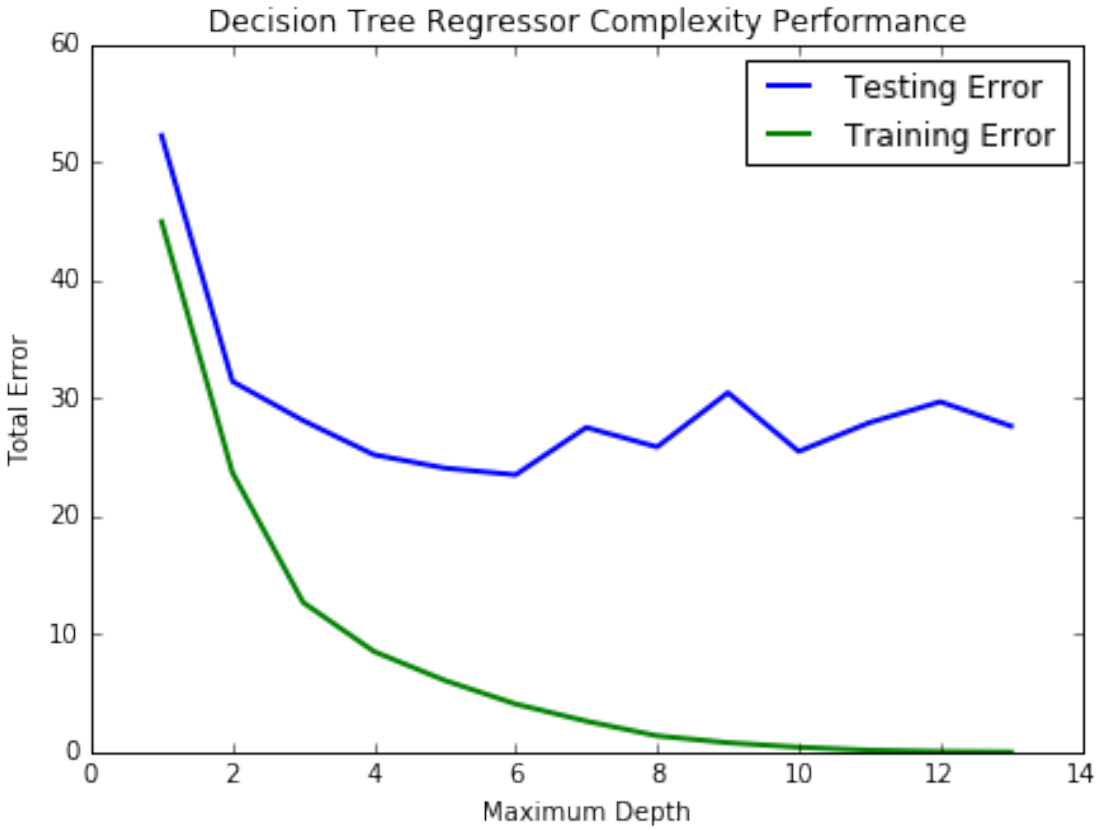
*Look at the learning curve graphs for the model with a max depth of 1 and a max depth of 10. When the model is using the full training set, does it suffer from high bias or high variance when the max depth is 1? What about when the max depth is 10?*

**Answer: it suffers from high bias when the max depth is 1, while suffers from high variance when the max depth is 10. When the max depth is 1, the training error and the testing error are both very high(about,while at the full size max depth 3 6 10 are lower than 30) at the end of the curve, so it may mainly suffer from the high-bias error which is not influenced by whether the data is training or testing data. When the max depth is 10, the training error and testing error has certain difference at the end of the curve and the training error at full size is almost zero, which is the feature of overfitting and high variance: the new coming data performs much worse than the training data.**

```
In [30]: model_complexity(X_train, y_train, X_test, y_test)

Creating a model complexity graph. . .
44.9726872904
52.265274531
23.697567073
31.4246885545
12.690436934
28.1091616034
8.53315122323
25.2075663706
6.08283428208
24.0791033011
4.07990155031
23.5074023597
2.62738745508
27.5231351809
1.38433800626
25.8674768149
0.799187982725
30.4734712805
0.417463911716
25.4908514996
0.16510791633
27.9346215511
0.0551713319637
```

```
29.6996547752
0.00783400591875
27.6332891261
```




### 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: As the max depth increases, training error reduces fast at the beginning and converged to a small value near zero at the end while testing error reduces fast at the beginning and converged to a certatin error at about 6 in x-axis. I think the max depth 6 may best generalizes the datasets because at this point, the testing curve gets its lowest and the training error is lower than 10 and its really acceptable.**

## 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 [80]: print "Final model has an optimal max_depth parameter of", reg.get_params
```

```
Final model has an optimal max_depth parameter of 7
```

**Answer: The optimal max_depth parameter for my model is 4, it's a little smaller than my initial intuition.**

### 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 [88]: 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: 21.630
```

**Answer: The best selling price for my client's home is 21.630. It's near the mean housing price and the median housing price.**

### 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 would use this model to predict the selling price of future clients. From learning curve its training error and testing error are acceptable. Additionally, I got different results between 4 and 8 when I ran the function fit_model. I tried 10 times and got 4 more often than others. And the predict answer with the parameter 4 got the answer near the median housing price so I think it's got a higher probability to be the best prediciton. By the way, the grid search in the datasets raise my confidence for the algorithm and I think it's a general method to predict selling price in the place not far from Boston area because the feature won't change too much. Of course the decision tree method isn't the only method, but it's easy and convenient for this kind of problem**

In [ ]: