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 [7]:
```
# 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 pr
ice
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 [8]: # 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 = 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: There are 13 attributes for each data point. Of these, the following seem important:

1. CRIM - per capita crime rate by town: Crime rate measures the number of crimes happening per unit of population in a given town. Intuitively, it makes sense that towns with lower crime rate are likely to have higher prices.
2. DIS - weighted distance to five Boston employment centers: The weighted distance to employment centers measures how much a person would have to commute to his/her place of work. It would seem reasonable to assume that if the house was close to all the employment centers, it would be pricier.
3. PTRATIO - Pupil-teacher ratio by town: This measures the number of students per teacher in schools in that area. A lesser PTRatio would mean students are getting more individual attention. Therefore, areas with a higher concentration of teachers and schools might be higher priced.

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 [9]:
```
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, 12.1
3]]
```

Answer:

Feature	Description	Value
CRIM	Per capita crime rate	11.95
DIS	Weighted distance to employment centers	1.385
PTRATIO	Pupil-teacher ratio	20.2

Evaluating Model Performance

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

Step 2

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

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

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

```
In [10]: # 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. """

             X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,
         random_state=0)

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


         # Test shuffle_split_data
         try:
             X_train, y_train, X_test, y_test = shuffle_split_data(housing_features, ho
         using_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:

Splitting our dataset into training and testing subsets helps us decide how evaluate the accuracy of the model on the test set. The performance on the test set is an indicator of how well our model is likely to perform on real and previously unseen data. Splitting it into train and test sets also helps us establish if our model is biased, underfit or overfit.

Without a training set, we would have no clear idea about how well the model actually performs.

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 [11]:
```
# Put any import statements you need for this code block here
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

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

    error = mean_absolute_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:

I used mean absolute error as the performance metric for analyzing the total error. As this is a regression problem, accuracy, precision, recall and F1 score cannot be used to evaluate performance. **Mean Squared Error (MSE)** gives us the square of the differences between actual and predicted values. **Mean Absolute Error (MAE)** measures the absolute difference (modulus of the differences) between actual and predicted values. While both performance metrics are reliable, MAE gives us a better idea of what the actual error is (without squaring it). Moreover, MSE is biased towards higher differences. Even if some predicted values are "way off" the mark, the MSE will be higher because of squaring the differences.

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 [12]:
```
# 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(performance_metric, greater_is_better=False
)

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

    # Fit the learner to the data to obtain the optimal model with tuned param
eters
    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: A grid search algorithm is useful for running our machine learning algorithm across different parameters and return the best combination of parameters so that our model has the least error. In the case above, we run our `GridSearchCV` across the parameters `{'max_depth':(1,2,3,4,5,6,7,8,9,10)}`. We could additionally keep our parameters `{'max_depth':(1,2,3,4,5,6,7,8,9,10), 'max_features':('auto', 'sqrt', 'log2')}` for a more extensive search.

The GridSearch algorithm is applicable when we do not know what the best combination of parameters would be for an accurate solution.

Question 6

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

Answer: Cross validation is a method of splitting a dataset into training and testing subsets while using all data points. In a *traditional train-test split*, the test data is never used in training. In a *k-fold cross validation*, the dataset is divided into *k* bins. The training algorithm is run *k* times. Through each iteration, one of the bins becomes the test set and the remaining *k-1* bins are used for training. This way every data point has been used for training as well as testing. The performance metric is averaged across all the *k* iterations and reported as the final performance.

Grid search uses a cross validation mechanism to suggest what are the best values for the parameters specified. By default, scikit-learn uses a 3-fold cross validation mechanism in `GridSearchCV`. As GridSearch goes through each of the parameters, it gives an output of the average performance across *k*-folds.

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 [15]: def learning_curves(X_train, y_train, X_test, y_test):
             """ Calculates the performance of several models with varying sizes of tra
         ining 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 ma
         x_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()
```

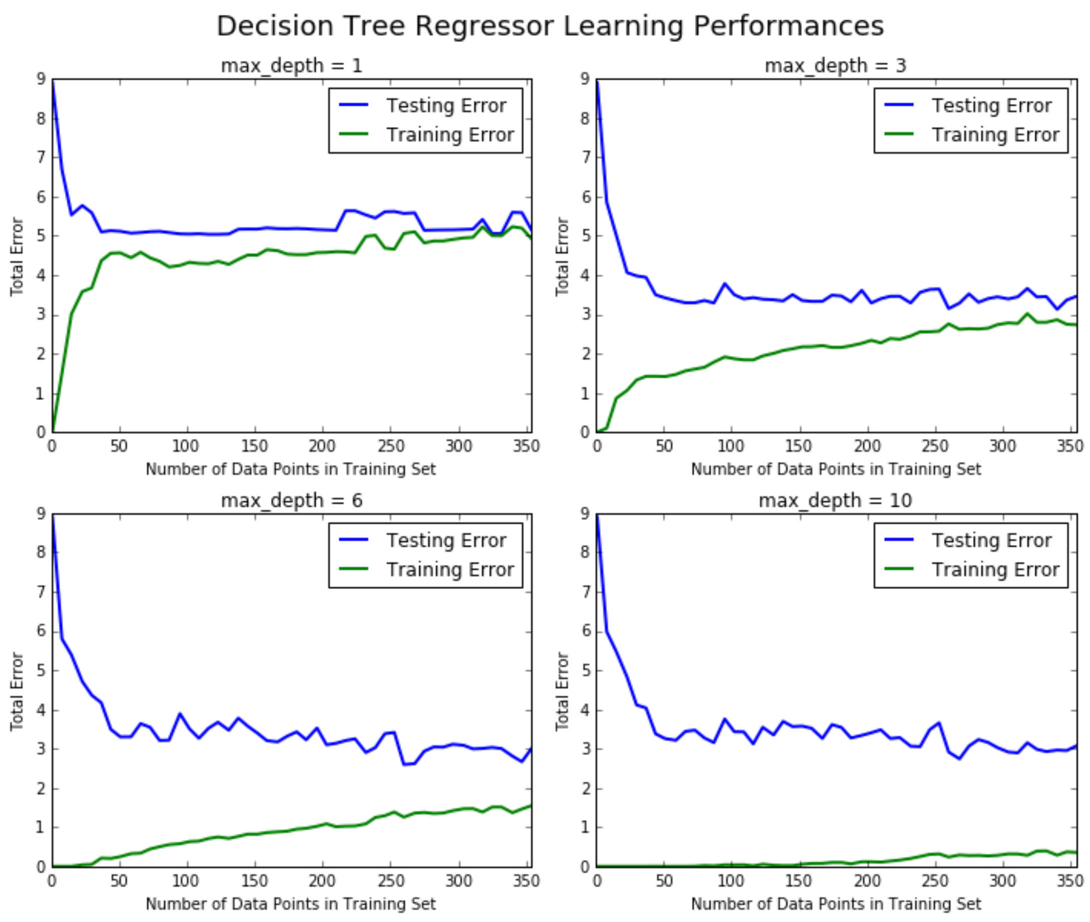
In [35]:
```
def model_complexity(X_train, y_train, X_test, y_test):
    """ Calculates the performance of the model as model complexity increases.
        The learning and testing errors rates are then plotted. """

    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 [17]:
```
learning_curves(X_train, y_train, X_test, y_test)
```

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



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: The *max depth* of my chosen model is 6. As the size of the training set increases, the training error increases. The testing error however decreases till about 50 data points quite exponentially. However, beyond the first 50 data points, the drop in testing error is quite small. This means that training a model with more than 50 data points is not very helpful.

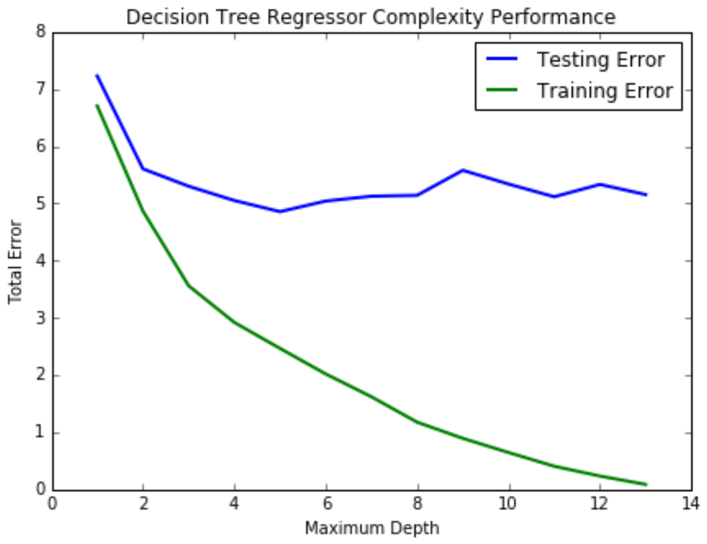
Question 8

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: Consider the model when the max depth is 1. When the model is using the full training set there is high bias. This is evident from the fact that the testing error and training error have converged. As we increase the depth, we see that the training error and testing error still converge, however they converge after more data points have been used. As we increase the depth further, the training error and testing error grow divergent. Moreover, the difference between the two errors also increases substantially. This a sign of overfitting. Consider the model wehere the max depth is 10. The plot for this model shows high variance. The most optimum model (somewhere between 4 to 6) have a small difference between the training and testing error and neither converge nor diverge. Those models therefore are optimally complex - neither underfit nor overfit.

```
In [37]: 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: From the graph, a maximum depth of 5 minimizes the total testing error. Therefore, the model with *max_depth = 5* generalizes the dataset the best. In the first stage of the graph, both the training and testing error are exponentially decreasing as we increase the maximum depth. However, in the second stage of the graph, the training error decreases however, the testing error remains more or less constant with increasing depth. This indicates that we are making the model more complex but we do not achieve any boost in accuracy (Occam's Razor). Therefore, we do not need to make the model more complex than required.

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

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

Answer: According to `GridSearchCV`, the final model has an optimal depth of 4. Intuitively it appeared that 5 should the optimum depth. However, the graph does not show standard deviation of the errors and this could be a cause for choosing an optimum depth of 4.

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

```
Predicted value of client's home: 20.720
```

Answer: Predicted value of client's home: 20.720

The statistics calculated earlier indicate that - Mean house price: 22.533 Median house price: 21.2

The selling price falls very close to the mean and median costs.

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, this model can be used to predict the selling price of future clients' homes in Greater Boston area. Based on parameter tuning, and the models that we made, the mean absolute error of approximately 2.432. This means that we could generally claim that our results would be correct within a range of 5 (thousand dollars) centered around our predicted value. Assuming this range is accurate enough, this model can be used to predict house prices.