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 [12]:
```
# 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, 332.09, 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 [13]: # Number of houses in the dataset
         total_houses = housing_features.shape[0]

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

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

         # Maximum housing value in the dataset
         maximum_price = 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: CRIM - crime rate is an important factor one should consider when buying a house, AGE - it is always better to live in a newly built buildings to avoid incidents, ZN - having grounds to have a breath of fresh air is

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: 11.91,90.0,0.0

Evaluating Model Performance

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

Step 2

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

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

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

```
In [15]: # 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,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, 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: We are using the test set to rate the performance of our model. The training set is used to train the model. Compared to training and testing our model on the original dataset (without splitting) This gives us a way to rate how well we are performing on out of sample data thus it allows us to avoid overfitting/underfitting the data.

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 [16]:
```
# 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 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 Exception as e:
    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: Accuracy, precision, recall and F1 Score are performance metrics used for classification. This leaves us with MAE and MSE. MSE will put more emphasis on the outliers , where as MAE will be less emphasising.MSE makes it easier to determine which is the best max_depth parameter to choose from the graph. MSE is a quadratic function which means it is differentiatable - we are able to find the minimum and the maximum easily.

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 [17]: # 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=scoring_function)

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

             # Return the optimal model
             t = reg.best_estimator_
             return t;


         # 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: It is an algorihm that selects the best parameters for the classifier by using K-fold cross validation. A better fit for larger dataset could be the RandomizedSearchCV algorithm. The Grid Search algorithm performs K-Fold cross validation on the dataset using each combination of parameters.Cross validation is used to rate the performance of the model by taking the model which has the lowest avarage error on all K trials performed by the Cross validation algorithm. It is worth mentioning that it is useful to tweak the parameter of the splits count used by cross validation, this will give us a more precise information about the performance of the solution at the cost of higher computing resources.

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 which splits the data in k folds. It is Each of these folds becomes the test set and the rest (k - 1) folds are used as the training set. Each point gets to be in the testing set exactly once and in the training set exactly k - 1 times. It is useful when using grid search because cross-validation helps us to find the parameters which result in the best performance on out of sample data by looking for the parameters which have the lowest avarage error on all K trials of cross-validation.

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 [18]:
```
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_dep
th = depth
            regressor = DecisionTreeRegressor(max_depth = depth)

            # Fit the learner to the training data
            regressor.fit(X_train[:s], y_train[:s])

            y_train.reshape(1,-1)
            # Find the performance on the training set
            train_err[i] = performance_metric(y_train[:s], regressor.predict(X_trai
n[:s]))

            y_test.reshape(1,-1)
            # 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()
```

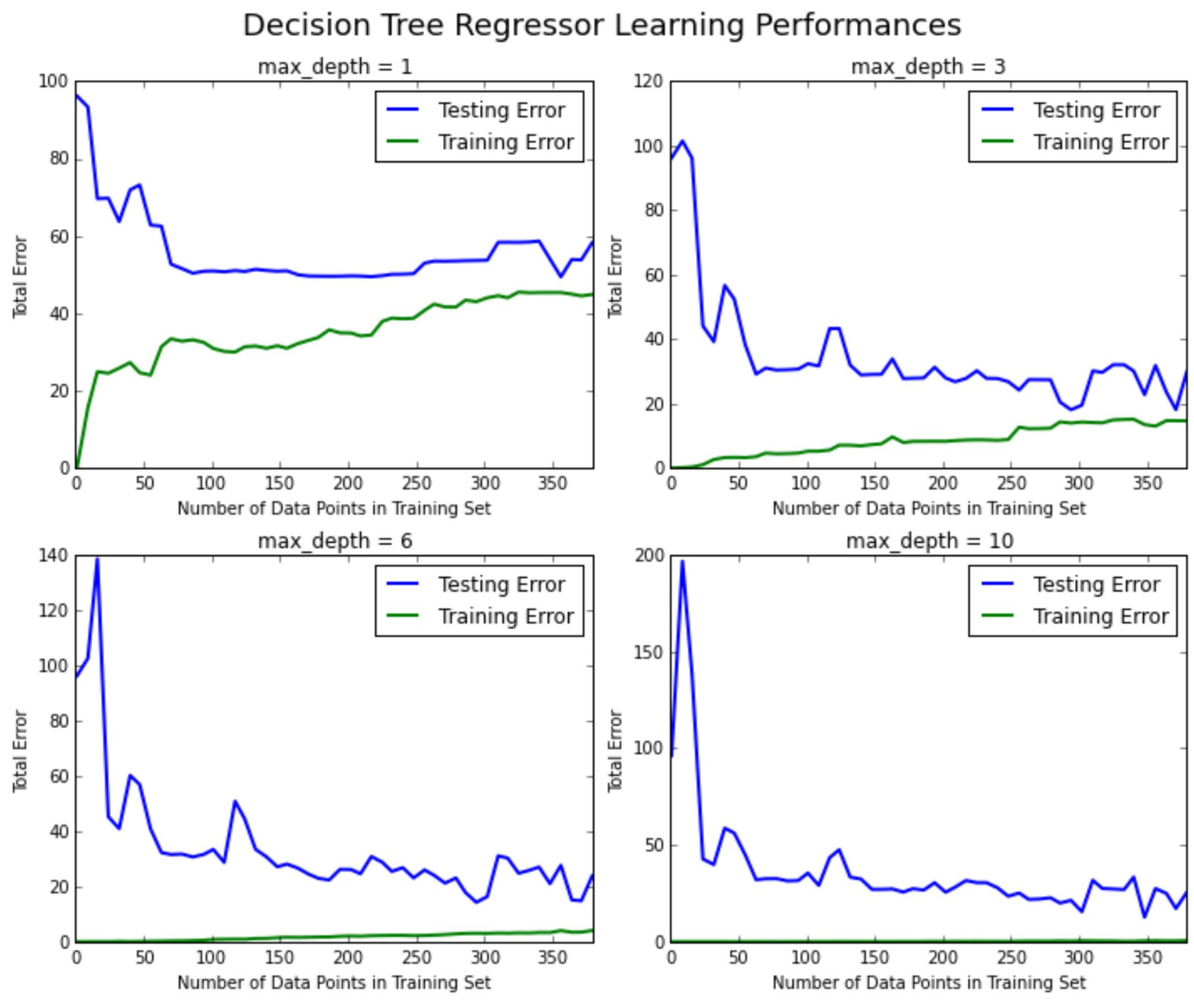
In [19]:
```
def model_complexity(X_train, y_train, X_test, y_test):
    """ Calculates the performance of the model as model complexity increases.
        The learning and testing errors rates are then plotted. """

    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 [20]: 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: Max depth is 6. The testing error decreases until the 50 points mark is approached. After that we see a spike in error which means that the model was unable to handle this particular chunk of data. After the spike at around 75 data points the test set error continues to decrease slowly which means that the model we have generalizes well out of sample data. The training error increases very slightly as the data points increase. This means that there exists some bias because we haven't perfectly fit the training set data which is good as we can see that the testing error is moving around 20 points thus we successfully handle out of sample data. This means that we are close to the optimal model for our problem - Low bias and low variance.

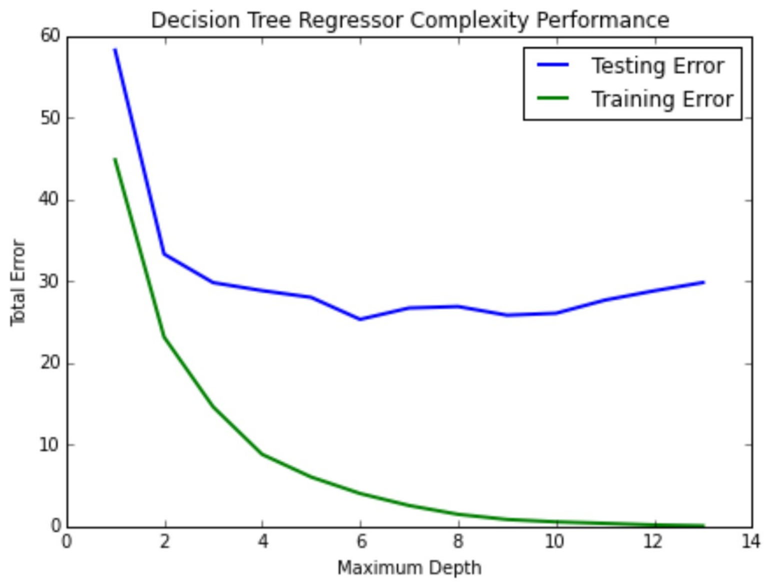
Question 8

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 max_depth is 1 - it suffers from high bias and the model will fail to generalize on out of sample cases. This can be determined by looking at the high error for Max_depth = 1 - around 50 points of error for the test set and around 45 for the training set which is far from the optimal. When Max depth = 10 it has big variance since it has almost no error on the training set it will overfit the new data given to it. This can be seen at the graph - almost no error on the training set and higher than the optimal error on the test set.

```
In [21]: model_complexity(X_train, y_train, X_test, y_test)
```

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



Question 9

From the model complexity graph above, describe the training and testing errors as the max depth increases. Based on your interpretation of the graph, which max depth results in a model that best generalizes the dataset? Why?

Answer: As the depth increases the model gets better at modeling the training set. The value when the algorithm will generalize the most is found when max_depth = 6 since it performs very well enough on the training set which means it is trained well and it also performs well on the test set. Thus it is the point where the bias and the variance are at their lowest values. We can see that in the initial stage the model performs poorly on both of the data sets. When max_depth reaches 6 this is where the bias and the variance are at their lowest (variance starts out low and increases as the complexity of the model increases). max_depth > 6 is when the variance begins to increase since we can see an increase in the test set error while the training set error is decreasing. This means that the model is starting to overfit. Thus when max_depth == 6 then we are at our lowest variance and lowest bias - our optimal model.

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

```
Final model has an optimal max_depth parameter of 5
```

Answer: If the grid search algorithm is used on the entire dataset the optimal max_depth parameter for my model is 5. It varies depending on the split and shuffle that the data recieved. This means that the max_depth parameter is somewhat biased by the way the data was split and shuffled. I think the best way to fix this is to add more data. My intution was that it will be 6 since it seemed that the classifier was trained the best at that point but that of course was on the training set and test set split from each other. The difference in the results somewhat expected since there is more data for the algorithm to process and this is the reason why the max_depth parameter is different from my intuition.

Question 11

With your parameter-tuned model, what is the best selling price for your client's home? How does this selling price compare to the basic statistics you calculated on the dataset?

Hint: Run the code block below to have your parameter-tuned model make a prediction on the client's home.

```
In [26]: 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: 20.968
```

Answer: The best selling price is 20.968 .It seems as it is a good prediction given that the house is in a dangerous neighbourhood thus should be slightly less expensive then the median. (It is in one standart diviation away from 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: The model has a low error rate this means that it is highly accurate and it is usable in predicting the prices of the homes in Greater Boston area in the year of 1993. The data is out of date for the current standarts of living thus I consider it unusable at the current point in time (The current median for house in Boston is around 400 000 $)

```
In [ ]:
```