Machine Learning Engineer Nanodegree

Model Evaluation & Validation

Project: Predicting Boston Housing Prices

Welcome to the first project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'Implementation'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Getting Started

In this project, you will evaluate the performance and predictive power of a model that has been trained and tested on data collected from homes in suburbs of Boston, Massachusetts. A model trained on this data that is seen as a *good fit* could then be used to make certain predictions about a home — in particular, its monetary value. This model would prove to be invaluable for someone like a real estate agent who could make use of such information on a daily basis.

The dataset for this project originates from the UCI Machine Learning Repository. The Boston housing data was collected in 1978 and each of the 506 entries represent aggregated data about 14 features for homes from various suburbs in Boston, Massachusetts. For the purposes of this project, the following preprocessing steps have been made to the dataset:

- 16 data points have an `'MEDV'` value of 50.0. These data points likely contain **missing or censored values** and have been removed.
- 1 data point has an `'RM'` value of 8.78. This data point can be considered an **outlier** and has been removed.
- The features `'RM'`, `'LSTAT'`, `'PTRATIO'`, and `'MEDV'` are essential. The remaining **non-relevant features** have been excluded.
- The feature `'MEDV'` has been **multiplicatively scaled** to account for 35 years of market inflation.

Run the code cell below to load the Boston housing dataset, along with a few of the necessary Python libraries required for this project. You will know the dataset loaded successfully if the size of the dataset is reported.

In [1]:

```
# Import libraries necessary for this project
import warnings; warnings.simplefilter('ignore')
import numpy as np
import pandas as pd
from sklearn.model_selection import ShuffleSplit

# Import supplementary visualizations code visuals.py
import visuals as vs

# Pretty display for notebooks
%matplotlib inline

# Load the Boston housing dataset
data = pd.read_csv('housing.csv')
prices = data['MEDV']
features = data.drop('MEDV', axis = 1)

# Success
print "Boston housing dataset has {} data points with {} variables each.".f
ormat(*data.shape)
```

```
Boston housing dataset has 489 data points with 4 variables each.
```

Data Exploration

In this first section of this project, you will make a cursory investigation about the Boston housing data and provide your observations. Familiarizing yourself with the data through an explorative process is a fundamental practice to help you better understand and justify your results.

Since the main goal of this project is to construct a working model which has the capability of predicting the value of houses, we will need to separate the dataset into **features** and the **target variable**. The **features**, `'RM'`, `'LSTAT'`, and `'PTRATIO'`, give us quantitative information about each data point. The **target variable**, `'MEDV'`, will be the variable we seek to predict. These are stored in `features` and `prices`, respectively.

In [2]:

```
print(features.describe())
```

```
               RM       LSTAT     PTRATIO
count  489.000000  489.000000  489.000000
mean     6.240288   12.939632   18.516564
std      0.643650    7.081990    2.111268
min      3.561000    1.980000   12.600000
25%      5.880000    7.370000   17.400000
50%      6.185000   11.690000   19.100000
75%      6.575000   17.120000   20.200000
max      8.398000   37.970000   22.000000
```

In [3]:

```
print(prices.describe())
```

```
count    4.890000e+02
mean     4.543429e+05
std      1.653403e+05
min      1.050000e+05
25%      3.507000e+05
50%      4.389000e+05
75%      5.187000e+05
max      1.024800e+06
Name: MEDV, dtype: float64
```

Implementation: Calculate Statistics

For your very first coding implementation, you will calculate descriptive statistics about the Boston housing prices. Since `numpy` has already been imported for you, use this library to perform the necessary calculations. These statistics will be extremely important later on to analyze various prediction results from the constructed model.

In the code cell below, you will need to implement the following:

- Calculate the minimum, maximum, mean, median, and standard deviation of `'MEDV'`, which is stored in `prices`.
  - Store each calculation in their respective variable.

In [4]:

```
# TODO: Minimum price of the data
minimum_price = np.min(prices)

# TODO: Maximum price of the data
maximum_price = np.max(prices)

# TODO: Mean price of the data
mean_price = np.mean(prices)

# TODO: Median price of the data
median_price = np.median(prices)

# TODO: Standard deviation of prices of the data
std_price = np.std(prices)

# Show the calculated statistics
print "Statistics for Boston housing dataset:\n"
print "Minimum price: ${:,.2f}".format(minimum_price)
print "Maximum price: ${:,.2f}".format(maximum_price)
print "Mean price: ${:,.2f}".format(mean_price)
print "Median price ${:,.2f}".format(median_price)
print "Standard deviation of prices: ${:,.2f}".format(std_price)
```

```
Statistics for Boston housing dataset:

Minimum price: $105,000.00
Maximum price: $1,024,800.00
Mean price: $454,342.94
Median price $438,900.00
Standard deviation of prices: $165,171.13
```

Question 1 - Feature Observation

As a reminder, we are using three features from the Boston housing dataset: `'RM'`, `'LSTAT'`, and `'PTRATIO'`. For each data point (neighborhood):

- `'RM'` is the average number of rooms among homes in the neighborhood.
- `'LSTAT'` is the percentage of homeowners in the neighborhood considered "lower class" (working poor).
- `'PTRATIO'` is the ratio of students to teachers in primary and secondary schools in the neighborhood.

*Using your intuition, for each of the three features above, do you think that an increase in the value of that feature would lead to an **increase** in the value of `'MEDV'` or a **decrease** in the value of `'MEDV'`? Justify your answer for each.*
Hint: Would you expect a home that has an `'RM'` value of 6 be worth more or less than a home that has an `'RM'` value of 7?

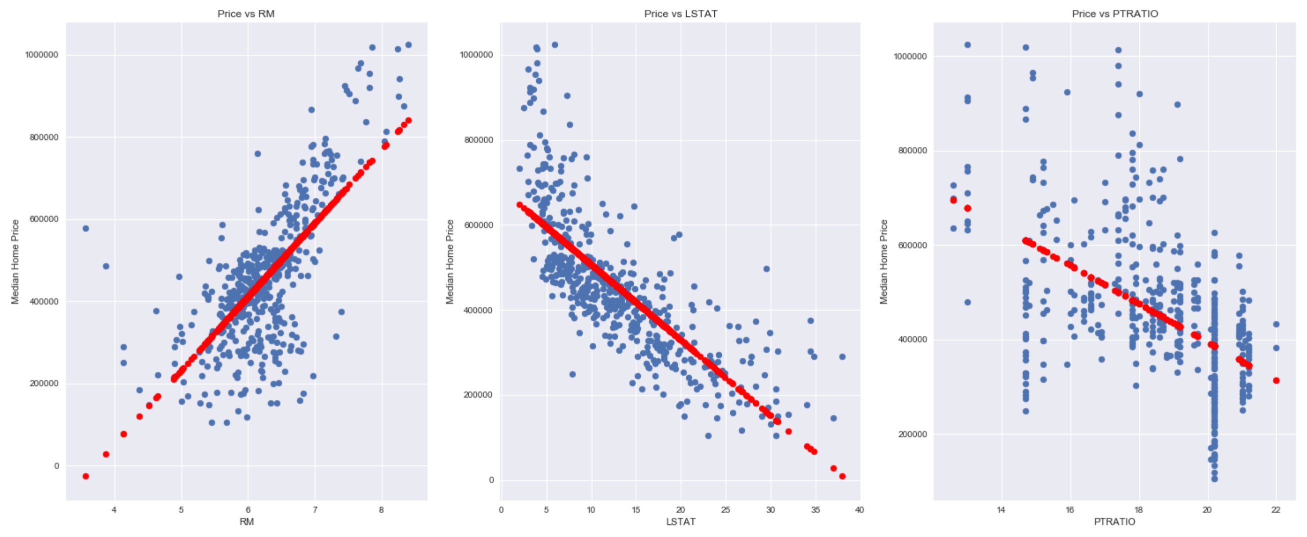
Answer: I would expect that "RM", or the average number of rooms among homes in the neighborhood would be positively correlated with median home prices. As the number of rooms in a house increases, the size of the house generally increases as well, and larger houses are more expensive because they require more material to construct and take up a larger plot of land. I would expect that "LSTAT", or the percentage of homeowners in the neighborhood considered "lower class", would be negatively correlated with median home prices. Neighborhoods with higher percentages of the working poor have lower median home prices because the citizens are not able to afford more expensive homes. Unfortunately, housing communities in the United States tend to be extremely segregated in terms of wealth, and less expensive homes, and thus poorer citizens, tend to cluster together. Finally, I would expect that "PTRATIO", or the ratio of students to teachers in primary and secondary schools in the neigborhood would be negatively correlated with median housing prices. A lower student:teacher ratio is generally viewed as a marker of a better school, and better schools typically lead to higher home prices in the surrounding neighborhoods. TO summarize my hypotheses: as RM increases, MEDV increases; as LSTAT increases, MEDV decreases; and as PTRATIO increases, MEDV decreases. These trends are demonstrated in the graph below.

In [5]:

```
% matplotlib inline

import matplotlib.pyplot as plt
import seaborn

plt.figure(figsize=(25,10))
for i, column in enumerate(features.columns):
    plt.subplot(1, 3, i+1)
    x = features[column].tolist()
    y = prices
    plt.scatter(x, y)
    z = np.polyfit(x, y, 1)
    p = np.poly1d(z)
    plt.scatter(x, p(x), c= 'r')
    plt.title("Price vs " + column)
    plt.xlabel(column)
    plt.ylabel('Median Home Price')
```



Developing a Model

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

Implementation: Define a Performance Metric

It is difficult to measure the quality of a given model without quantifying its performance over training and testing. This is typically done using some type of performance metric, whether it is through calculating some type of error, the goodness of fit, or some other useful measurement. For this project, you will be calculating the *coefficient of determination*, R^2, to quantify your model's performance. The coefficient of determination for a model is a useful statistic in regression analysis, as it often describes how "good" that model is at making predictions.

The values for R^2 range from 0 to 1, which captures the percentage of squared correlation between the predicted and actual values of the **target variable**. A model with an R^2 of 0 is no better than a model that always predicts the *mean* of the target variable, whereas a model with an R^2 of 1 perfectly predicts the target variable. Any value between 0 and 1 indicates what percentage of the target variable, using this model, can be explained by the **features**. *A model can be given a negative R^2 as well, which indicates that the model is **arbitrarily worse** than one that always predicts the mean of the target variable.*

For the `performance_metric` function in the code cell below, you will need to implement the following:

- Use `r2_score` from `sklearn.metrics` to perform a performance calculation between `y_true` and `y_predict`.
- Assign the performance score to the `score` variable.

In [6]:

```
from sklearn.metrics import r2_score
```

```
def performance_metric(y_true, y_predict):
    """ Calculates and returns the performance score between
        true and predicted values based on the metric chosen. """

    score = r2_score(y_true, y_predict)

    # Return the score
    return score
```

Question 2 - Goodness of Fit

Assume that a dataset contains five data points and a model made the following predictions for the target variable:

True Value	Prediction
3.0	2.5
-0.5	0.0
2.0	2.1
7.0	7.8
4.2	5.3

Would you consider this model to have successfully captured the variation of the target variable? Why or why not?

Run the code cell below to use the `performance_metric` function and calculate this model's coefficient of determination.

```
In [7]:
```

```
# Calculate the performance of this model
score = performance_metric([3, -0.5, 2, 7, 4.2], [2.5, 0.0, 2.1, 7.8, 5.3])
print "Model has a coefficient of determination, R^2, of {:.3f}.".format(sc
ore)
```

```
Model has a coefficient of determination, R^2, of 0.923.
```

Answer: Yes, I would consider this model to have captured the variation of the target variable. As shown by the R^2 (coefficent of determination) value of 0.923, the linear model closely predicts the target values. The R^2 value is the percentage of the dependent variable that is explained by a linear model. In this example, the R^2 value demonstrates that the model accounts for 92.3% of the variability in the target values.

Implementation: Shuffle and Split Data

Your next implementation requires that you take the Boston housing dataset and split the data into training and testing subsets. Typically, the data is also shuffled into a random order when creating the training and testing subsets to remove any bias in the ordering of the dataset.

For the code cell below, you will need to implement the following:

- Use `train_test_split` from `sklearn.cross_validation` to shuffle and split the

`features` and `prices` data into training and testing sets.
  - Split the data into 80% training and 20% testing.
  - Set the `random_state` for `train_test_split` to a value of your choice. This ensures results are consistent.
- Assign the train and testing splits to `X_train`, `X_test`, `y_train`, and `y_test`.

In [8]:

```
from sklearn.model_selection import train_test_split

# There is no acceptable random state except for the answer to life, the un
iverse, and everything
X_train, X_test, y_train, y_test = train_test_split(features, prices,
test_size=0.2, random_state=42)

# Success
print "Training and testing split was successful."
```

```
Training and testing split was successful.
```

Question 3 - Training and Testing

What is the benefit to splitting a dataset into some ratio of training and testing subsets for a learning algorithm?
Hint: What could go wrong with not having a way to test your model?

Answer: Splitting a dataset into a training and testing set can help you avoid overfitting your model on the training set. The testing set is used to check if the model can generalize well to new instances. If a model performs well on the training set (achieves a high performance metric) but performs poorly on the testing set, then the model has likely been overfit to the training set and more data needs to be aquired and fed into the model, or the hyperparameters need to be tuned. A testing set is not always the perfect solution to the problem of over-fitting, but it serves as a useful check before a model is used to predict labels for new instances.

Analyzing Model Performance

In this third section of the project, you'll take a look at several models' learning and testing performances 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 performance. 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.

Learning Curves

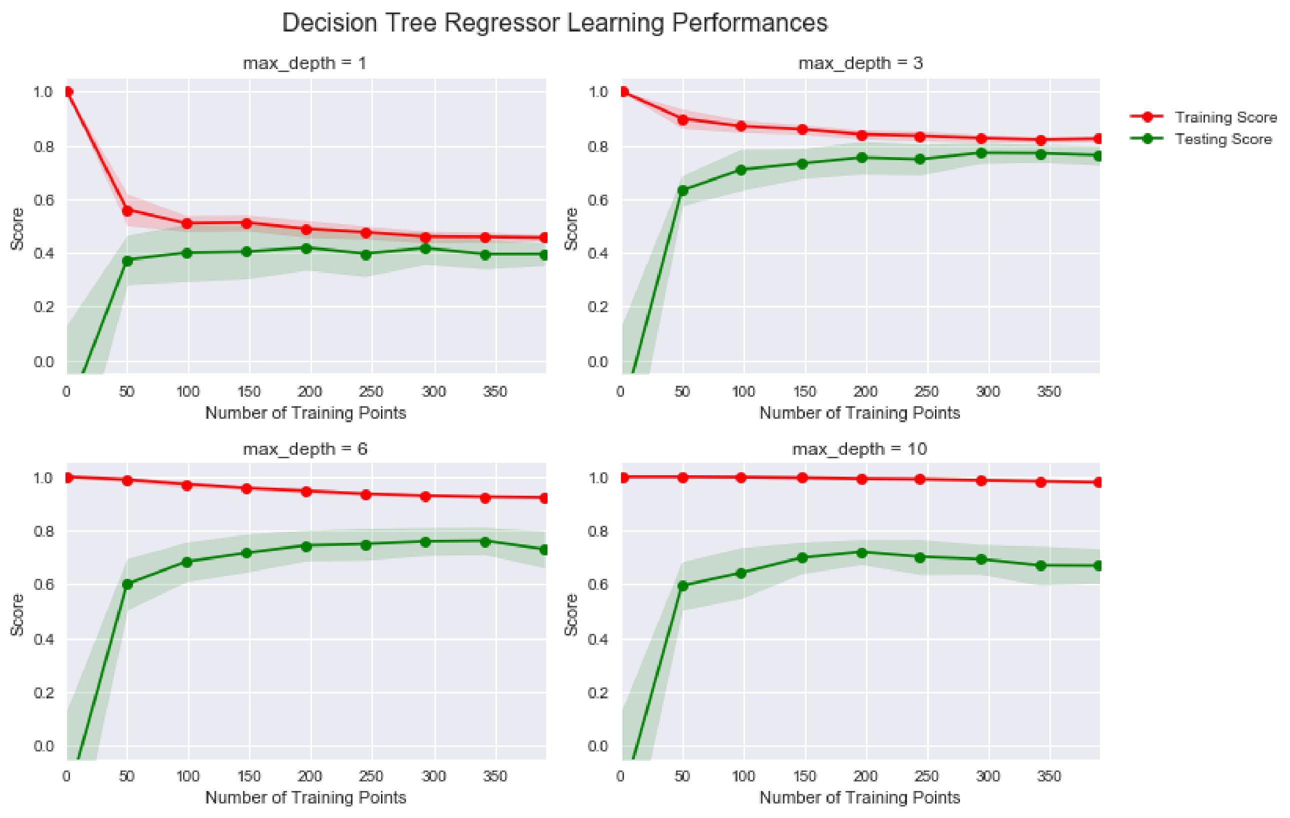
The following code cell produces four graphs for a decision tree model with different maximum depths. Each graph visualizes the learning curves of the model for both training and testing as the size of the training set is increased. Note that the shaded region of a learning curve denotes the uncertainty of that curve (measured as the standard deviation). The model is scored on both the training and testing sets using R^2, the coefficient of determination.

training and testing sets using R^2, the coefficient of determination.

Run the code cell below and use these graphs to answer the following question.

In [9]:

```
# Produce learning curves for varying training set sizes and maximum
depths
vs.ModelLearning(features, prices)
```



Question 4 - Learning the Data

Choose one of the graphs above and state the maximum depth for the model. What happens to the score of the training curve as more training points are added? What about the testing curve? Would having more training points benefit the model?
Hint: Are the learning curves converging to particular scores?

Answer: I chose the graph with maximum depth = 3. As the number of training points increases from 0 to 300 points, the performance on the training set (given here by the r2 score) decreases and the score on the testing set increases. However, after around 300 training points, the training score converges to a score slightly above 0.8, and the testing score converges to a score slightly below 0.8. This suggests there is a diminishing return to adding training data points. The initial increase in number of training points from 0 to 50 results in an increase in accuracy on the testing set from 0% to 65%, but from 50 training points to 300 training points, the accuracy on the testing set only increases to near 80%. This demonstrates that the initial 50 point increase in training data is much more effective in improving the testing performance of the model than subsequent increases. Another interesting aspect of the curve is that with fewer training points, the model is cleary overfitting the training data. With a limited amount of training data, the model is simply memorizing the training data which explains the seemingly extraordinary performance on the training set. The corresponding low score on the testing set shows that the overfit model can not generalize well to new cases. As the amount of training data is increased, the training and testing scores approach the same value indicating that the model is no longer overfitting the training data and that the model can adequately indicating that the model is no longer overfitting the training data and that the model can adequately generalize to new instances. To summarize, increasing the amount of training data increases the performance of the model by decreasing the extent of overfitting up to a certain quantity of data, at which point the improvement diminishes and the training and testing scores begin to converge to a single value.

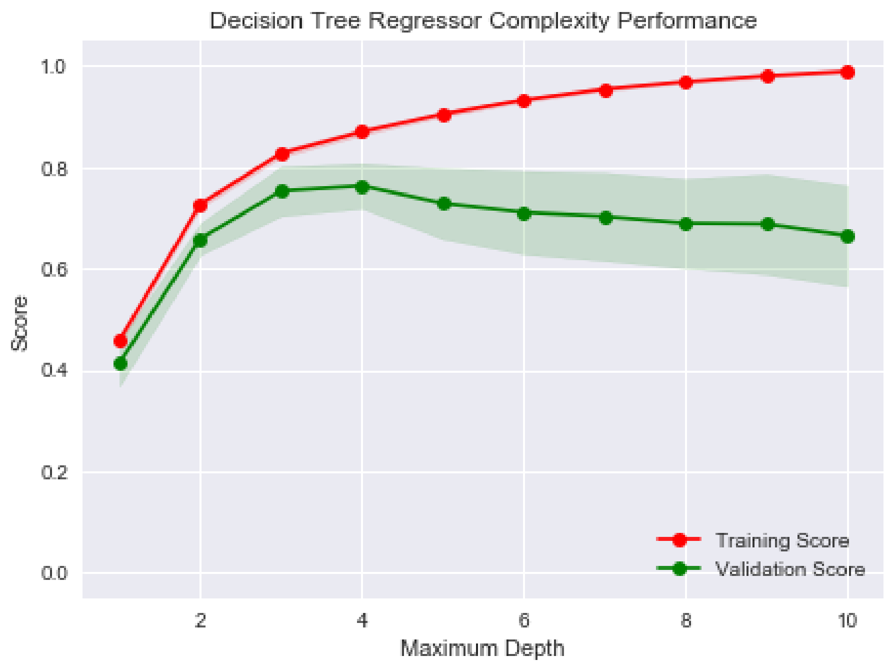
Complexity Curves

The following code cell produces a graph for a decision tree model that has been trained and validated on the training data using different maximum depths. The graph produces two complexity curves — one for training and one for validation. Similar to the **learning curves**, the shaded regions of both the complexity curves denote the uncertainty in those curves, and the model is scored on both the training and validation sets using the `performance_metric` function.

Run the code cell below and use this graph to answer the following two questions.

In [10]:

```
vs.ModelComplexity(X_train, y_train)
```



Question 5 - Bias-Variance Tradeoff

When the model is trained with a maximum depth of 1, does the model suffer from high bias or from high variance? How about when the model is trained with a maximum depth of 10? What visual cues in the graph justify your conclusions?
Hint: How do you know when a model is suffering from high bias or high variance?

Answer: When the decision tree model is trained with a maximum depth of 1, it suffers from high bias. This can be seen in the low score (as given by the r2 performance metric) on the training data. The model at this point is not complex enough to represent the underlying relationships in the data and is therefore not able to learn the trends in the data and make accurate predictions. As the complexity of the model increases (as the maximum depth is increased), the bias decreases and the variance increases. At a maximum depth of 10, the decision tree is now exhibiting high variance and overfitting to the training data. This is demonstrated by the high score on the training set and the low score on the testing set. The significant difference in training and validation scores is an indicator of high variance in the model. Above a maximum depth of 6, the model becomes too complex for the task, overfits the training data, and cannot generalize well to the new cases it encounters in the testing set.

Question 6 - Best-Guess Optimal Model

Which maximum depth do you think results in a model that best generalizes to unseen data? What intuition lead you to this answer?

Answer: The optimal maximum depth for the decision tree is 4 as this maximimizes the score on the testing set. I arrived at this conclusion from examining the validation curve and recording the maximum depth where the performance score was greatest for the testing set. At a maximum depth of 4, the decision tree achieves the best balance between bias and variance. The model is able to represent the underlying relationships in the data while avoiding the mistake of overfitting the training data.

Evaluating Model Performance

In this final section of the project, you will construct a model and make a prediction on the client's feature set using an optimized model from `fit_model`.

Question 7 - Grid Search

What is the grid search technique and how it can be applied to optimize a learning algorithm?

Answer: The grid search technique is a way to evaluate a large number of algorithm hyperparameters and choose the combination that results in the highest score on the chosen performance metric. It can be applied to optimize a learning algorithm by creating a grid of possible combinations for the algorithm, and then calling GridSearchCV in scikit-learn with the classifier (decision tree in this case), the parameter grid, and the scoring method (r2 in this case). GridSearchCV will then test all the parameter combinations and record the set of parameters that achieves the best score. This set of parameters can then be implemented in a final model. Although Grid Search can be computationally expensive, it is often more efficient than evaulating a number of parameter combinations manually.

Question 8 - Cross-Validation

What is the k-fold cross-validation training technique? What benefit does this technique provide for grid search when optimizing a model?
Hint: Much like the reasoning behind having a testing set, what could go wrong with using grid search without a cross-validated set?

Answer: K-Fold cross-validation is a method used to avoid overfitting a classifier to a particular training set. K-Fold cv splits the data into k subsets (called folds) on each iteration and trains the model using k-1 of these subsets. The final fold is held out as a test set and used to determine the performance of the trained model. For example, if k=10, then the training data will be split into 10 subsets, and each iteration, the model will be fitted (trained) to 9 of these sets and tested on the 10th set. A total of 10 iterations will be performed with the model is tested on a different fold on each iteration. The final step in the cross validation process is to average the performance metric across all of the iterations to arrive at a single score for assessing the model. K-Fold cv can be used in combination with grid search to ensure that each combination of parameters is thoroughly tested. If cv is not used in Grid Search, the model that returns the highest performance score may simply be the best suited to the particular training data and will not generalize well to new data. K-Fold cross validation can be used to correct this problem by ensuring that all of the parameter combinations are trained and tested multiple times on different sets of the training data. The result of using K-Fold cv together with Grid Search is that the highest scoring model will better suited to generalize to new data.

Implementation: Fitting a Model

Your final implementation requires that you bring everything together and train a model using the **decision tree algorithm**. To ensure that you are producing an optimized model, you will train the model using the grid search technique to optimize the `'max_depth'` parameter for the decision tree. The `'max_depth'` parameter can be thought of as how many questions the decision tree algorithm is allowed to ask about the data before making a prediction. Decision trees are part of a class of algorithms called *supervised learning algorithms*.

In addition, you will find your implementation is using `ShuffleSplit()` for an alternative form of cross-validation (see the `'cv_sets'` variable). While it is not the K-Fold cross-validation technique you describe in **Question 8**, this type of cross-validation technique is just as useful!. The `ShuffleSplit()` implementation below will create 10 (`'n_splits'`) shuffled sets, and for each shuffle, 20% (`'test_size'`) of the data will be used as the *validation set*. While you're working on your implementation, think about the contrasts and similarities it has to the K-fold cross-validation technique.

Please note that ShuffleSplit has different parameters in scikit-learn versions 0.17 and 0.18. For the `fit_model` function in the code cell below, you will need to implement the following:

- Use `DecisionTreeRegressor` from `sklearn.tree` to create a decision tree regressor object.
  - Assign this object to the `'regressor'` variable.
- Create a dictionary for `'max_depth'` with the values from 1 to 10, and assign this to the `'params'` variable.
- Use `make_scorer` from `sklearn.metrics` to create a scoring function object.
  - Pass the `performance_metric` function as a parameter to the object.
  - Assign this scoring function to the `'scoring_fnc'` variable.
- Use `GridSearchCV` from `sklearn.grid_search` to create a grid search object.
  - Pass the variables `'regressor'`, `'params'`, `'scoring_fnc'`, and `'cv_sets'` as parameters to the object.
  - Assign the `GridSearchCV` object to the `'grid'` variable.

In [11]:

```
from sklearn.metrics import make_scorer
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV

def fit_model(X, y):
    """ Performs grid search over the 'max_depth' parameter for a 
        decision tree regressor trained on the input data [X, y]. """
    
    # Create cross-validation sets from the training data
    # sklearn version 0.18: ShuffleSplit(n_splits=10, test_size=0.1, train_
size=None, random_state=None)
    # sklearn versiin 0.17: ShuffleSplit(n, n_iter=10, test_size=0.1, train
_size=None, random_state=None)
    cv_sets = ShuffleSplit(n_splits = 10, test_size = 0.20, random_state = 
0)

    # TODO: Create a decision tree regressor object
    regressor = DecisionTreeRegressor(random_state=42)

    # TODO: Create a dictionary for the parameter 'max_depth' with a range 
from 1 to 10
    params = {"max_depth": np.arange(1,11).tolist()}

    # TODO: Transform 'performance_metric' into a scoring function using 'm
ake_scorer' 
    scoring_fnc = make_scorer(performance_metric)

    # TODO: Create the grid search object
    grid = GridSearchCV(regressor, params, scoring=scoring_fnc, cv=cv_sets)

    # Fit the grid search object to the data to compute the optimal model
    grid = grid.fit(X, y)

    # Return the optimal model after fitting the data
    print "The best score was {:.4f} for a Decision Tree with a max depth o
f {}.".format(grid.best_score_, grid.best_params_['max_depth'])
    return grid.best_estimator_
```

Making Predictions

Once a model has been trained on a given set of data, it can now be used to make predictions on new sets of input data. In the case of a *decision tree regressor*, the model has learned *what the best questions to ask about the input data are*, and can respond with a prediction for the **target variable**. You can use these predictions to gain information about data where the value of the target variable is unknown — such as data the model was not trained on.

Question 9 - Optimal Model

*What maximum depth does the optimal model have? How does this result compare to your guess in **Question 6**?*

Run the code block below to fit the decision tree regressor to the training data and produce an optimal model.

In [12]:

```
# Fit the training data to the model using grid search
reg = fit_model(X_train, y_train)
```

```
# Produce the value for 'max_depth'
print "Parameter 'max_depth' is {} for the optimal model.".format(reg.get_p
arams()['max_depth'])
```

```
The best score was 0.7623 for a Decision Tree with a max depth of 4.
Parameter 'max_depth' is 4 for the optimal model.
```

Answer: The optimal maximum depth for the Decision Tree Regressor is **4**. This matches the answer I gave for Question 6.

Question 10 - Predicting Selling Prices

Imagine that you were a real estate agent in the Boston area looking to use this model to help price homes owned by your clients that they wish to sell. You have collected the following information from three of your clients:

Feature	Client 1	Client 2	Client 3
Total number of rooms in home	5 rooms	4 rooms	8 rooms
Neighborhood poverty level (as %)	17%	32%	3%
Student-teacher ratio of nearby schools	15-to-1	22-to-1	12-to-1

What price would you recommend each client sell his/her home at? Do these prices seem reasonable given the values for the respective features?
Hint: Use the statistics you calculated in the **Data Exploration** section to help justify your response.

Run the code block below to have your optimized model make predictions for each client's home.

```
In [13]:
```

```
# Produce a matrix for client data
client_data = [[5, 17, 15], # Client 1
               [4, 32, 22], # Client 2
               [8, 3, 12]]  # Client 3

# Show predictions
for i, price in enumerate(reg.predict(client_data)):
    print "Predicted selling price for Client {}'s home: ${:,.2f}".format(i
+1, price)
```

```
Predicted selling price for Client 1's home: $403,025.00
Predicted selling price for Client 2's home: $237,478.72
Predicted selling price for Client 3's home: $931,636.36
```

Answer: I would recommend that my clients put their houses up for sale at the following prices: Client 1: **$403,025**, Client 2: **$237,479**, Client 3: **$931,636**.

These prices match the hypotheses I formulated in the data exploration section. The model predicts that Client 3's house should be the most expensive which agrees with my intuition as this house is in the upper quartile for number of rooms, in the bottom quartile for percentage of working poor citizens in the neighborhood, and in the bottom quartile for student-to-teacher ratio for schools in the neighborhood. I therefore would predict that this house should have a value near the top of all houses in Boston, which is what the model returns. Client 1's house is closer to the mean than Client 3 in terms of rooms, percentage of citizens considered lower status, and the student to teacher ratio in surrounding schools. I would predict this house has a value very close to the median home price in all of Boston, which is in agreement with the model prediction. Finally, Client 2's house has a low number of rooms, is in a neighborhood with a relatively high level of poverty, and is in a neighborhood with a high student to teacher ratio and consequently should have the lowest value of the three houses. This is what the model predicts as it places the value of the house in the bottom quartile of all houses in Boston. All of the predicted prices are reasonable considering the features of the homes in relation to all the houses in Boston. The graph below shows the predicted prices of the client's house in relation to all of the houses in Boston and the first quartile, median, and third quartile of median housing prices in Boston.

In [14]:

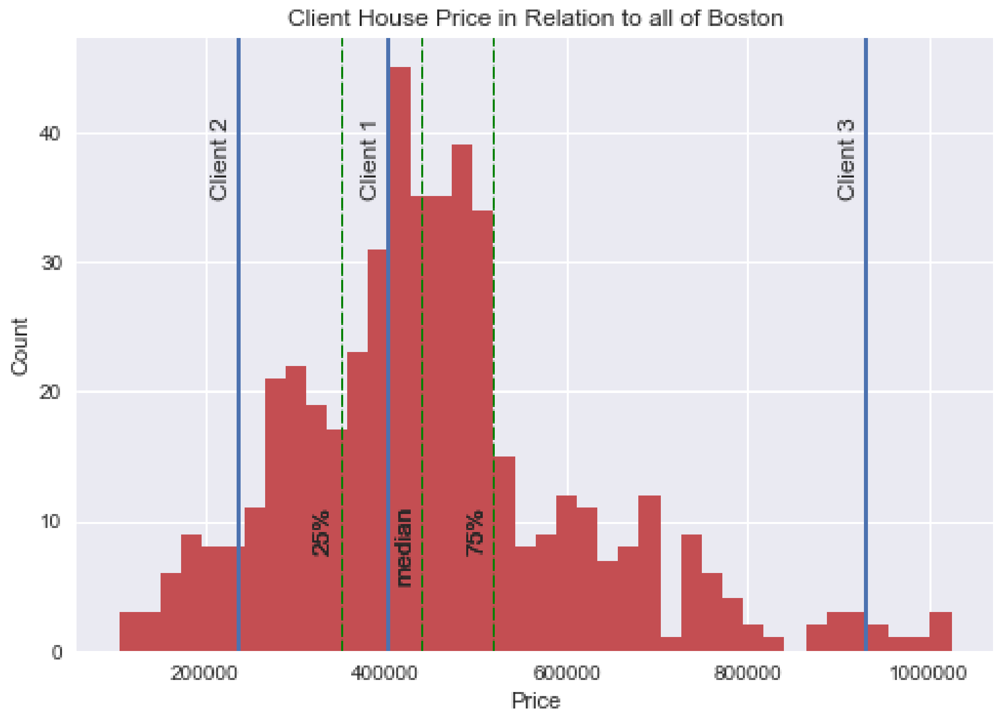
```
first_quartile = prices.describe()['25%']
median_price = np.median(prices)
third_quartile = prices.describe()['75%']
```

In [15]:

```
for i, price in enumerate(reg.predict(client_data)):
    plt.hist(prices, bins = 40)
    plt.axvline(price, lw = 2)
    plt.text(price-30000, 40, 'Client '+str(i+1), rotation = 90)

    plt.axvline(first_quartile, lw=1, c='g', ls='--')
    plt.text(first_quartile-30000, 10, '25%', rotation=90)
    plt.axvline(median_price, lw=1, c='g', ls='--')
    plt.text(median_price-30000, 10, 'median', rotation=90)
    plt.axvline(third_quartile, lw=1, c='g', ls='--')
    plt.text(third_quartile-30000, 10, '75%', rotation=90)

    plt.title('Client House Price in Relation to all of Boston')
    plt.ylabel('Count')
    plt.xlabel('Price')
```



Sensitivity

An optimal model is not necessarily a robust model. Sometimes, a model is either too complex or too simple to sufficiently generalize to new data. Sometimes, a model could use a learning algorithm that is not appropriate for the structure of the data given. Other times, the data itself could be too noisy or contain too few samples to allow a model to adequately capture the target variable — i.e., the model is underfitted. Run the code cell below to run the `fit_model` function ten times with different training and testing sets to see how the prediction for a specific client changes with the data it's trained on.

In [16]:

```
vs.PredictTrials(features, prices, fit_model, client_data)
```

```
The best score was 0.7754 for a Decision Tree with a max depth of 4.
Trial 1: $391,183.33
The best score was 0.7809 for a Decision Tree with a max depth of 5.
Trial 2: $419,700.00
The best score was 0.7264 for a Decision Tree with a max depth of 4.
Trial 3: $415,800.00
The best score was 0.7329 for a Decision Tree with a max depth of 4.
Trial 4: $420,622.22
The best score was 0.7442 for a Decision Tree with a max depth of 5.
Trial 5: $418,377.27
The best score was 0.7358 for a Decision Tree with a max depth of 4.
Trial 6: $411,931.58
The best score was 0.8074 for a Decision Tree with a max depth of 4.
Trial 7: $399,663.16
The best score was 0.7854 for a Decision Tree with a max depth of 4.
Trial 8: $407,232.00
The best score was 0.7160 for a Decision Tree with a max depth of 3.
Trial 9: $351,577.61
The best score was 0.8187 for a Decision Tree with a max depth of 4.
Trial 10: $413,700.00

Range in prices: $69,044.61
```

Question 11 - Applicability

In a few sentences, discuss whether the constructed model should or should not be used in a real-world setting.
Hint: Some questions to answering:

- *How relevant today is data that was collected from 1978?*
- *Are the features present in the data sufficient to describe a home?*
- *Is the model robust enough to make consistent predictions?*
- *Would data collected in an urban city like Boston be applicable in a rural city?*

Answer: The decision tree house pricing model should not be implemented in a real-world setting. There are a number of problems with both the data fed into the model and the classifier itself. The data is relatively old and is likely to not be relevant for predicting home prices in 2017. Moreover, the data used in the classifier was composed of only three features, and there may be additional features publically available that are more predictive of housing prices. The value that consumers place on certain home features could also have evolved over the years which would require updating the features fed into the model for training and predicting. In order to create a model that could be successfully implemented today, I would need to have access to recent data with more features describing the houses. Furthermore, the model itself has an issue of variability. The price predicted by the model for a house with identical features differed by as much as $69,000. This may not seem like the model for a house with identical features differed by as much as $69,000. This may not seem like much to some investors, but it is a serious amount of money to the average homeowner in the United States as it is more than the average American household makes in a year. Moreover, the model likely would not generalize well to different cities or regions that have different levels of poverty, different education levels, and different average house sizes. To summarize, I would only rely on this model for pricing houses in Boston in 1978, and even then, I would be skeptical of the predictions. There have been housing price prediction models that perform well, but nearly all of these are based on a specific geographic region for a specific time frame. It would be difficult to create a model that could accurately predict home prices across the entire United States because the underlying relationships are different in every city, and a model would not be able to account for all the inherent variability. The model developed in this project is impressive at predicting prices on a narrow subset of Boston housing data, but it is best suited for a teaching application using historical data and should not be implemented in the real world.

> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to
> **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.