#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 [40]:
```
# 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 notebo
%matplotlib inline

# Create our client's feature set for which we will be predicting a se
CLIENT_FEATURES = [[11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.38

# 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 [41]:
```
# Number of houses in the dataset
# print housing_features
total_houses = len(housing_prices)

# Number of features in the dataset
total_features = len(housing_features[0])

# 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 = 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 - The amount of crimes commited divided by number of residents, in each town. This would give an idea of the attractiveness of the neighbourhood. RM - The average (probably mean) number of rooms in each house in the area. This would give a rough indication of the size of the houses. RAD - A measurement of how accessible the radial highways are from the house. This gives an approximation of how convenient the houses are for traveling to other places.

##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 [42]:
```
import pandas as pd

df = pd.DataFrame(CLIENT_FEATURES, columns = city_data.feature_names)
print df[['CRIM', 'RM', 'RAD']]
print CLIENT_FEATURES
```

```
    CRIM     RM  RAD
0  11.95  5.609   24
[[11.95, 0.0, 18.1, 0, 0.659, 5.609, 90.0, 1.385, 24, 680.0, 20.2, 3
32.09, 12.13]]
```

Answer: CRIM - 11.95 RM - 5.609 RAD - 24

#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 if the `shuffle_split_data` function is working if the statement "*Successfully shuffled and split the data!*" is printed.

In [43]:
```
# 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 subs
        then returns the training and testing subsets. """

    # Shuffle and split the data
    X_train, X_test, y_train, y_test = cross_validation.train_test_spli

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


# Test shuffle_split_data
try:
    X_train, X_test, y_train, y_test = shuffle_split_data(housing_featu
    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 4 *Why do we split the data into training and testing subsets for our model?*

Answer: Because without testing we have no way of knowing or validating how well our model generalises to new data. We split out data into a separate testing set as if we tested on data we had previously trained on we would be liable to overfit. This means the model would fit the training data really well but wouldn't be able to generalise to new data in future.

##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 if the `performance_metric` function is working if the statement "*Successfully performed a metric calculation!*" is printed.

In [81]:
```
# Put any import statements you need for this code block here
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error

def performance_metric(y_true, y_predict):
    """ Calculates and returns the total error between true and predict
        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:
    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: As we are creating a regression model (it has a continuous output) the accuracy, precision, recall and so F1 Score are all unsuitable as we care more about how far the prediction is from the actual value than how many predictions the model gets exactly right.

From MSE and MAE I would chose MSE as it is less prone to ambiguity (absolute error can have the same total value for different regressions, presuming the split of the data is the same), as well as penalising predicted values that are further from the real values more.

This is weighed up against the effect outliers have on both metrics. As MSE penalises greater error more this means it is less robust to outliers and so the regression model created will be affected negatively to a greater degree should we have outliers in our data.

##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 if the `fit_model` function is working if the statement "*Successfully fit a model to the data!*" is printed.

In [122]:
```
# Put any import statements you need for this code block
from sklearn.grid_search import GridSearchCV
from sklearn.grid_search import RandomizedSearchCV
from sklearn.metrics import make_scorer
from sklearn.decomposition import PCA

def fit_model(X, y):
    """ Tunes a decision tree regressor model using GridSearchCV on the
        and target labels y and returns this optimal model. """

    # Create a decision tree regressor object
    clf = DecisionTreeRegressor()

    # Set up the parameters we wish to tune
    param_dist = {'max_depth':(3,4,5,6,7,8,9,None), 'max_features':(2,4
                  'min_samples_split':(2,3,4,5,6,7,8)}

    # Make an appropriate scoring function
    scoring_function = make_scorer(mean_squared_error, greater_is_bette

    # Make the GridSearchCV object
    reg = RandomizedSearchCV(clf, param_distributions=param_dist, scori

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

    # Return the optimal model
    return reg


# Test fit_model on entire dataset


try:
    reg = fit_model(X_train, y_train)
    print "Successfully fit a model!"
    print "Final model optimal parameters:", reg.best_params_, performa
    print performance_metric(y_train, reg.predict(X_train))
except:
    print "Something went wrong with fitting a model."
```

```
Successfully fit a model!
Final model optimal parameters: {'min_samples_split': 8, 'max_featur
es': 11, 'max_depth': 4} 18.6028850402
9.02016791253
```

##Question 5 *What is the grid search algorithm and when is it applicable?*

Answer: The grid search algorithm enables the testing of multiple combinations of parameters for the model to detect the combination that gives the best score on the cross validation sets using the chosen scoring method.

It is applicable where it is unclear the best set of parameters to use for a model, so in testing all combinations of parameters the best model can be found.

##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 splits the training data into x sections, one of which will be taken out before training and this will then be used as a form of testing set for the model. This process will repeat x amount of times for the amount of sections. Repetition allows all the data to used as testing and training as different sections get selected as the testing set in each run. This repetition also reduces the possibility of an unfavourable or favourable split influencing the performance disproportionately. The results from this testing on the cross validation sets will be used in this case (using Grid Search) to select the best parameters to use.

The cross validation set is distinct from the testing set as the testing set will only be touched once at the end of the training. Had the testing set been used for cross validation this opens the model up to over fitting as the testing data will no longer be unseen and so is being used to optimise the model, selecting which parameters to use. Future generalisation to new data may be compromised.

It is useful when using Grid Search as it will give a more accurate way of measuring the performance of each set of model parameters allowing us to test their ability to fit to data not trained on.

I have used RandomisedSearchCV here as it strikes a balance between allowing a wide range of parameters with which to select the best model, and building the model in the smallest amount of time. GridSearchCV due to conducting an exhaustive search can be very time intensive.

#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 [123]:
```
def learning_curves(X_train, y_train, X_test, y_test):
    """ Calculates the performance of several models with varying sizes
        The learning and testing error rates for each model are then pl

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

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

    # We will vary the training set size so that we have 50 different s
    sizes = np.round(np.linspace(1, len(X_train), 50))
    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
            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.pr

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

        # 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', fonts
    fig.tight_layout()
    fig.show()
```

In [124]:
```
def model_complexity(X_train, y_train, X_test, y_test):
    """ Calculates the performance of the model as model complexity inc
        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 wit
        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_

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

    # 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 [125]:
```
learning_curves(X_train, y_train, X_test, y_test)
```

```
/Users/WonderWaffle/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:24: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
/Users/WonderWaffle/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:27: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
/Users/WonderWaffle/anaconda/lib/python2.7/site-packages/matplotlib/figure.py:397: UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the figure
  "matplotlib is currently using a non-GUI backend, "

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 3. As the size of the training set increases the training error increases, and the testing error sharply decreases until it levels off at around 28.

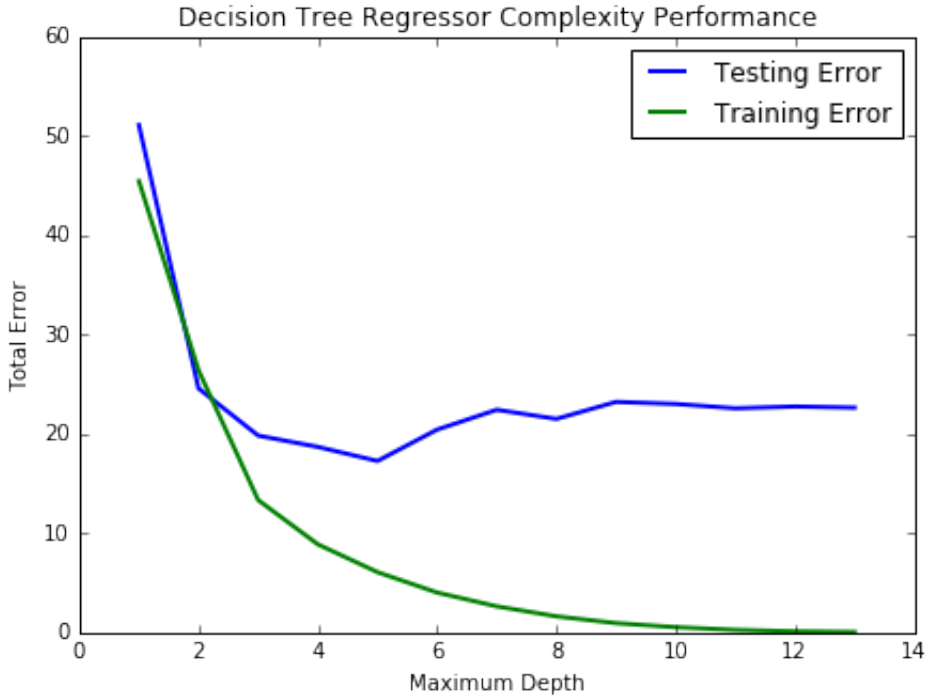
##Question 8 *Look at the learning curve graphs for the model with a max depth of 1 and a max depth of 10. When the model is using the full training set, does it suffer from high bias or high variance when the max depth is 1? What about when the max depth is 10?*

Answer: Max depth of 1 suffers from high bias, both testing and training error are large as the model ignores the data too much and doesn't fit it well.

Max depth of 10 suffers from high variance. Training error is minimal due to the model overfitting the training data, perfectly matching it. This means testing error is much higher than the training error due to the model not generalising to new data well.

In [126]:
```
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: ** The training error decreases and starts leveling off as total error reaches 0.

Testing error decreses sharply before levelling off at around 22.

Max depth 4 best generalises the dataset as testing error is near it's minimum for the graph implying bias isn't too high and the model performs well. As well as this the difference between training and testing error is minimal indicating that variance isn't high and the model isn't overfitting the data.
```

#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`. *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 [127]:
```
print "Final model optimal parameters:", reg.best_params_, performance
```

```
Final model optimal parameters: {'min_samples_split': 8, 'max_featur
es': 11, 'max_depth': 4} 18.6028850402
```

```
**Answer: ** Max depth is 4.

This is same as my intuition, demostrating that the model is
selecting a max depth that minimises difference between training and
testing errors, while minimising testing error.

An additional benefit of max depth 4 is it is a fairly simple
hypothesis, which occam's razor dictates is more preferential due to
being more likely to be the "true" hypothesis.
```

##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 [128]:
```
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: 21.004
```

Answer: 21.004

The calculated mean was 22.533, and the median 21.1. Standard deviation was 9.188.

The selling price calculated is fairly near the average calculated with either mean or median. The difference is also much smaller than the standard deviation, showing the value is not unusual within the dataset.

##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' homes in Greater Boston area. The value given for our client's home was believable based on metrics previously computed, and the mean square error was efficiently minimised via the tuning of the parameters.

In addition the values given on rerunning the code were consistent and similar each time.

One thing that hasn't been checked is outliers in the data; verification that these aren't present, and excluding them if they are, would give additional confidence of the models ability to generalise and to estimate the selling price of future clients' homes.
Outlier removal gives consistency to model selection due to lower risk of one of the sections throwing out the analysis due to presence of outliers (although cross validation with it's repetition should reduce this), as well as giving consistency to the result with a lower risk that the presence of outliers in the testing set will give an unreliable score.