MLE-P1

March 4, 2016

1 Machine Learning Engineer Nanodegree

1.1 Model Evaluation & Validation

1.2 Project 1: Predicting Boston Housing Prices

Welcome to the first project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been written. You will need to implement additional functionality to successfully answer all of the questions for this project. Unless it is requested, do not modify any of the code that has already been included. In this template code, there are four sections which you must complete to successfully produce a prediction with your model. Each section where you will write code is preceded by a **STEP X** header with comments describing what must be done. Please read the instructions carefully!

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

A description of the dataset can be found here, which is provided by the **UCI Machine Learning Repository**.

2 Getting Started

To familiarize yourself with an iPython Notebook, **try double clicking on this cell**. You will notice that the text changes so that all the formatting is removed. This allows you to make edits to the block of text you see here. This block of text (and mostly anything that's not code) is written using Markdown, which is a way to format text using headers, links, italics, and many other options! Whether you're editing a Markdown text block or a code block (like the one below), you can use the keyboard shortcut **Shift + Enter** or **Shift + Return** to execute the code or text block. In this case, it will show the formatted text.

Let's start by setting up some code we will need to get the rest of the project up and running. Use the keyboard shortcut mentioned above on the following code block to execute it. Alternatively, depending on your iPython Notebook program, you can press the **Play** button in the hotbar. You'll know the code block executes successfully if the message *"Boston Housing dataset loaded successfully!"* is printed.

```
In [1]: # Importing a few necessary libraries
        import numpy as np
        import matplotlib.pyplot as pl
        from sklearn import datasets
        from sklearn.tree import DecisionTreeRegressor

        # Make matplotlib show our plots inline (nicely formatted in the notebook)
        %matplotlib inline

        # Create our client's feature set for which we will be predicting a selling price
        CLIENT_FEATURES = [[11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385, 24, 680.0, 20.20, 332

        # 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!
```

3 Statistical Analysis and Data Exploration

In this first section of the project, you will quickly investigate a few basic statistics about the dataset you are working with. In addition, you'll look at the client's feature set in CLIENT_FEATURES and see how this particular sample relates to the features of the dataset. Familiarizing yourself with the data through an explorative process is a fundamental practice to help you better understand your results.

3.1 Step 1

In the code block below, use the imported numpy library to calculate the requested statistics. You will need to replace each None you find with the appropriate numpy coding for the proper statistic to be printed. Be sure to execute the code block each time to test if your implementation is working successfully. The print statements will show the statistics you calculate!

```
In [2]: import pandas as pd
        names = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax', 'ptratio', 'b'
        df = pd.DataFrame(housing_features, columns=names)

        # Number of houses in the dataset
        total_houses = df.shape[0]

        # Number of features in the dataset
        total_features = df.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
```

3.2 Question 1

As a reminder, you can view a description of the Boston Housing dataset here, where you can find the different features under **Attribute Information**. The `MEDV` attribute relates to the values stored in our `housing_prices` variable, so we do not consider that a feature of the data.

Of the features available for each data point, choose three that you feel are significant and give a brief description for each of what they measure.

Remember, you can **double click the text box below** to add your answer!

Answer:
CRIM: criminality in the area
NOX: polution levels
RM: number of rooms

3.3 Question 2

Using your client's feature set CLIENT_FEATURES, which values correspond with the features you've chosen above?
Hint: Run the code block below to see the client's data.

```
In [3]: 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:
CRIM: 11.95
NOX: 0.659
RM: 5.609

4 Evaluating Model Performance

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

4.1 Step 2

In the code block below, you will need to implement code so that the `shuffle_split_data` function does the following: - Randomly shuffle the input data `X` and target labels (housing values) `y`. - Split the data into training and testing subsets, holding 30% of the data for testing.

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

```
In [4]: 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  = 0.3, random_state=12

             # 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!
```

4.2 Question 4

Why do we split the data into training and testing subsets for our model?

Answer:

We split the data into training and testing because we want to use one part of the data (training) to learn from and the other part (testing) of the data to verify the performance of our model (how close to the truth our predictions are). If we don't separate the data and use all of it to train and to test, me may be getting the false impression of a good model because we'd be evaluating it on information already seem.

4.3 Step 3

In the code block below, you will need to implement code so that the `performance_metric` function does the following: - Perform a total error calculation between the true values of the `y` labels `y_true` and the predicted values of the `y` labels `y_predict`.

You will need to first choose an appropriate performance metric for this problem. See the sklearn metrics documentation to view a list of available metric functions. **Hint:** Look at the question below to see a list of the metrics that were covered in the supporting course for this project.

Once you have determined which metric you will use, remember to include the necessary import statement as well!
Ensure that you have executed the code block once you are done. You'll know if the `performance_metric` function is working if the statement *"Successfully performed a metric calculation!"* is printed.

```
In [5]: #from sklearn.metrics import mean_absolute_error
        from sklearn.metrics import mean_squared_error
```

```
#from sklearn.metrics import explained_variance_score
#from sklearn.metrics import median_absolute_error
#from sklearn.metrics import r2_score


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_absolute_error(y_true, y_predict)
    error = mean_squared_error(y_true, y_predict)
    #error = explained_variance_score(y_true, y_predict)
    #error = median_absolute_error(y_true, y_predict)
    #error = r2_score(y_true, y_predict)

    return error


# Test performance_metric
try:
    total_error = performance_metric(y_train, y_train)
    print "Successfully performed a metric calculation!"
except:
    print "Something went wrong with performing a metric calculation."
```

```
Successfully performed a metric calculation!
```

4.4 Question 4

Which performance metric below did you find was most appropriate for predicting housing prices and analyzing the total error. Why? - Accuracy - Precision - Recall - F1 Score - Mean Squared Error (MSE) - Mean Absolute Error (MAE)

Answer:

Mean squared error because it penalizes the error more than the absolute error. mean_squared_error(y_train[0:1],y_train[0:1]+2) returns 4 as mean_absolute_error(y_train[0:1],y_train[0:1]+2) returns 2

The other metrics are used for classification problems.

4.5 Step 4 (Final Step)

In the code block below, you will need to implement code so that the `fit_model` function does the following: - Create a scoring function using the same performance metric as in **Step 2**. See the sklearn `make_scorer` documentation. - Build a GridSearchCV object using `regressor`, `parameters`, and `scoring_function`. See the sklearn documentation on GridSearchCV.

When building the scoring function and GridSearchCV object, *be sure that you read the parameters documentation thoroughly.* It is not always the case that a default parameter for a function is the appropriate setting for the problem you are working on.

Since you are using `sklearn` functions, remember to include the necessary import statements below as well!

Ensure that you have executed the code block once you are done. You'll know if the `fit_model` function is working if the statement *"Successfully fit a model to the data!"* is printed.

```
In [17]: from sklearn.grid_search import GridSearchCV
         from sklearn.metrics import make_scorer
```

```
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(mean_squared_error)

    # Make the GridSearchCV object
    reg = GridSearchCV(regressor, parameters, scoring='mean_squared_error')

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

    # Return the optimal model
    return reg


# Test fit_model on entire dataset
#reg = fit_model(housing_features, housing_prices)
#reg.best_params_
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!
```

4.6 Question 5

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

Answer:

Grid Search loops through a pre-defined set of parameters (and its values) and finds the ones that perform best. It is aplicable wheneve we are looking to go through parameter combinations and optimize our models automatically.

4.7 Question 6

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

Answer:

As discussed before, splitting the data into training and testing is a crucial step but sometimes doing it only once is not enough because we may accidentally overfit our model if the training set is somehow imbalanced. Cross Validation splits the data into train and test X amount of times (X being what's called the number of folds) and trains the model on the training part and tests the model on the test part. It allows you to have several training + testing sets inside your actual training set. That way the parameters will be optimized on the entire data set and any random anomalies due to random splitting will be removed.

Also, it maximizes data usage. When a dataset is limited in size cross validation becomes extremely useful as it allows for an extensive exploitation of available data allowing assessing the real potential of our algorithm in terms of performance metrics.

5 Checkpoint!

You have now successfully completed your last code implementation section. Pat yourself on the back! All of your functions written above will be executed in the remaining sections below, and questions will be asked about various results for you to analyze. To prepare the **Analysis** and **Prediction** sections, you will need to intialize the two functions below. Remember, there's no need to implement any more code, so sit back and execute the code blocks! Some code comments are provided if you find yourself interested in the functionality.

```
In [7]: 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.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 with max_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 [8]: 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()
            return test_err
```

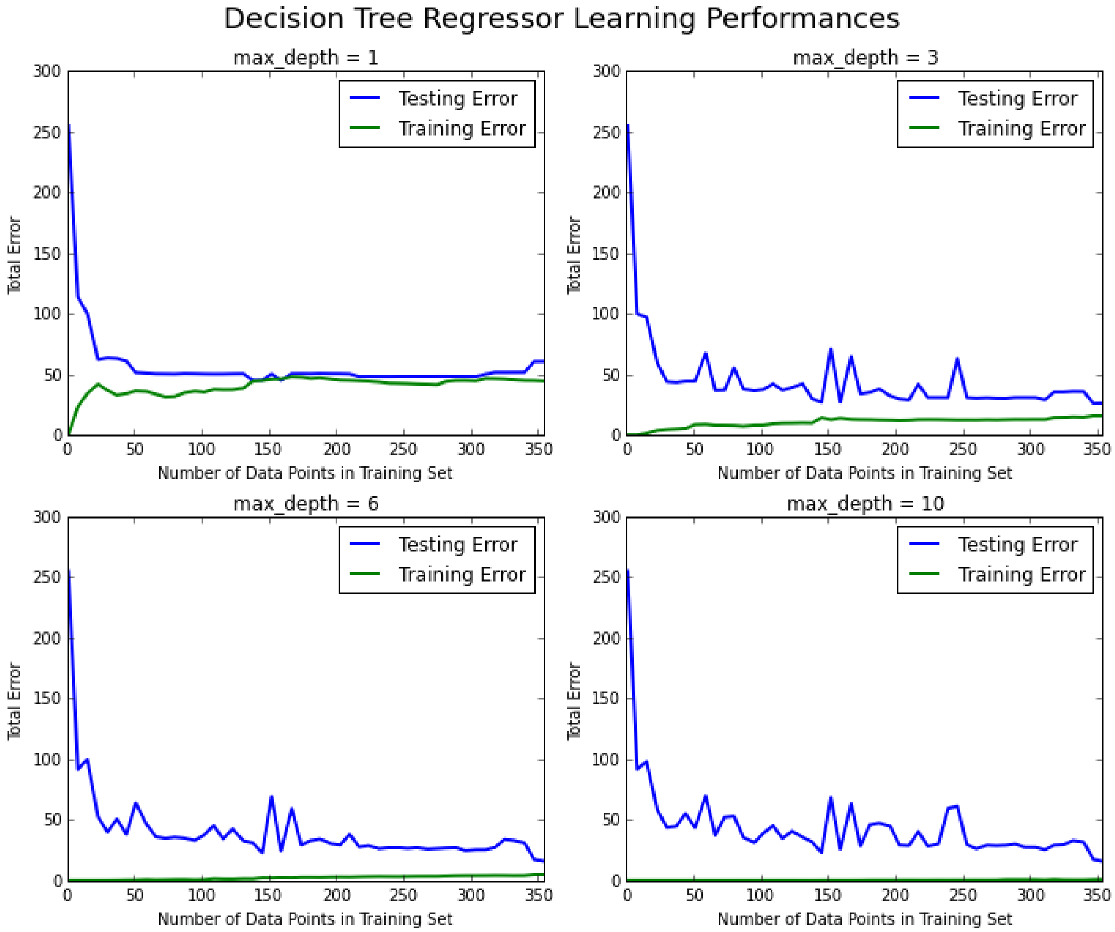
6 Analyzing Model Performance

In this third section of the project, you’ll take a look at several models’ learning and testing error rates on various subsets of training data. Additionally, you’ll investigate one particular algorithm with an increasing `max_depth` parameter on the full training set to observe how model complexity affects learning and testing errors. Graphing your model’s performance based on varying criteria can be beneficial in the analysis process, such as visualizing behavior that may not have been apparent from the results alone.

```
In [9]: learning_curves(X_train, y_train, X_test, y_test)
```

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

```
C:\Anaconda\lib\site-packages\matplotlib\figure.py:387: UserWarning: matplotlib is currently using a nor
  "matplotlib is currently using a non-GUI backend, "
```



6.1 Question 7

Choose one of the learning curve graphs that are created above. What is the max depth for the chosen model? As the size of the training set increases, what happens to the training error? What happens to the testing error?

Answer:

On the max_depth = 6, more training data sligtly increases the training error and decreases the test error. The rate the test error decreases gets smaller as the number of points increases until such point it starts to plateau.

6.2 Question 8

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

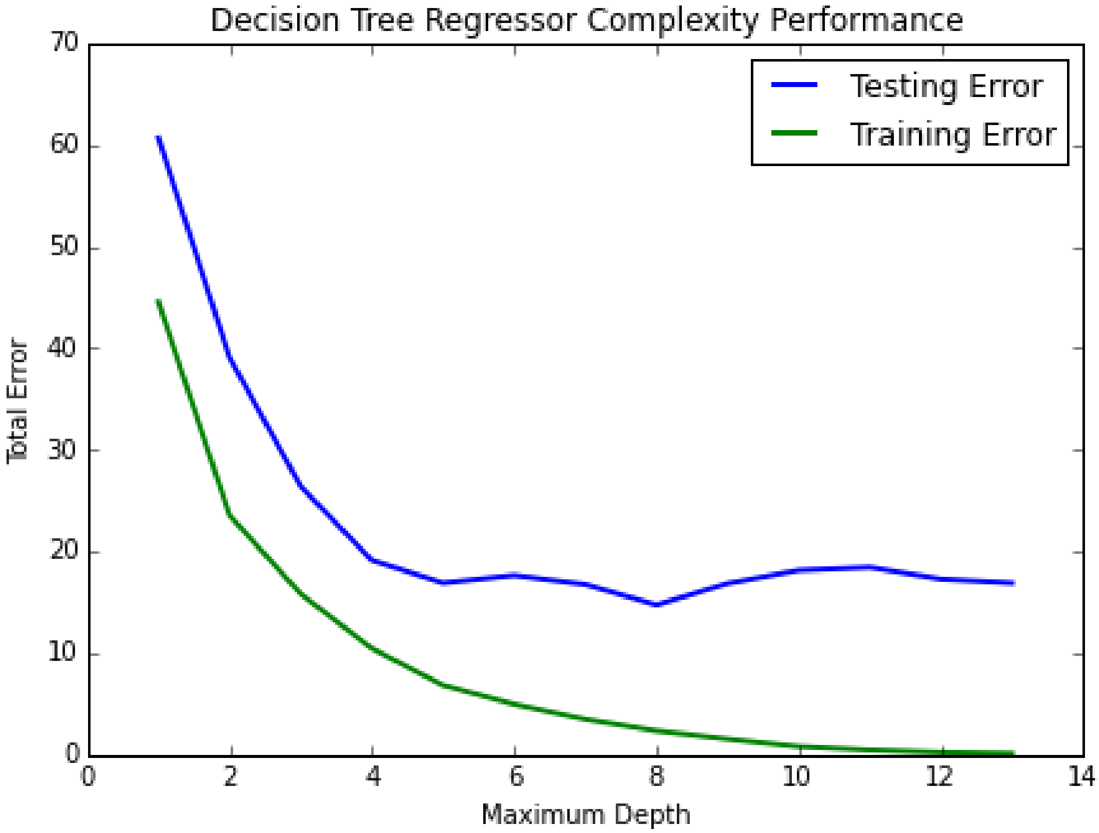
Answer:

On max depth = 1 it suffers from high bias because the training and testing errors converge and are quite high (no matter how much data we feed it, the model cannot represent the underlying relationship and therefore has systematic high errors).

On max depth = 10 it suffers from high variance because we have a large gap between the training and testing error. Unlike a bias model, models that suffer from variance can generally improve if we have more data to learn from or we can simplify the model representing the most important features of the data.

```
In [10]: test_err = model_complexity(X_train, y_train, X_test, y_test)
```

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



6.3 Question 9

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

Answer:

As the complexity (max depth) increases, the train data error decreases, whihc indicates that the model is overfiting. It isn't really learning anything new as it is "memorizing" all records of the training data. As if the tree model were building one branch of the tree for each observation.

The testing data, on the other hand, decreases for a while (there is clearly more thing to learn if you can build a tree with 3 levels than one with 2 levels) but it reachs a point (around 5 I'd say) that the extra branches don't really add any new information.

7 Model Prediction

In this final section of the project, you will make a prediction on the client's feature set using an optimized model from `fit_model`. *To answer the following questions, it is recommended that you run the code blocks* *several times and use the median or mean value of the results.*

7.1 Question 10

Using grid search on the entire dataset, what is the optimal **`max_depth`** *parameter for your model? How does this result compare to your intial intuition?*
Hint: Run the code block below to see the max depth produced by your optimized model.

```
In [11]: print "Final model optimal parameters:", reg.best_params_
```

```
Final model optimal parameters: {'max_depth': 4}
```

Answer:
The max_depth suggested is 4, which is close to what I had guessed above (5).

It is important to note that the max_depth from the “reg” object is obtained training and testing on the full data set and using 3 fold cross validation (default to GridSearchCV) and that, due to the nature of CV, if the code is executed twice, a different value can be found.

The max depth of my “initial intuition” (question 9) was concluded using the ” X_test” and “y_test” data set we split in the begining, so a difference between the values is expected.

7.2 Question 11

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

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

```
In [18]: sale_price = reg.predict(CLIENT_FEATURES)
         print "Predicted value of client's home: {0:.3f}".format(sale_price[0])
```

```
Predicted value of client's home: 21.630
```

```
In [19]: from sklearn.neighbors import NearestNeighbors
         def find_nearest_neighbor_indexes(x, X):  # x is your vector and X is the data set.
             neigh = NearestNeighbors(n_neighbors = 10 )
             neigh.fit(X)
             distance, indexes = neigh.kneighbors( x )
             return indexes

         indexes = find_nearest_neighbor_indexes(CLIENT_FEATURES, housing_features)
         sum_prices = []
         for i in indexes:
             sum_prices.append(city_data.target[i])
         neighbor_avg = np.mean(sum_prices)
         print "Nearest Neighbors average: " +str(neighbor_avg)
```

```
Nearest Neighbors average: 21.52
```

Answer:
It predicts 21.63 which is very close to the median (21.19) and the mean (22.53) so we know we are not dealing with a big outlier. Using a second method of prediction (Nearest Neighbors) also outputs a similar value than our tree prediction.

7.3 Question 12 (Final Question):

In a few sentences, discuss whether you would use this model or not to predict the selling price of future clients' homes in the Greater Boston area.

Answer:

That's actually a tough question. I would relly on the model, just always be aware that, even though we have a machine learning model, it is not perfect and we got an average mean square error of around 17, which represents 4 thousands dollars of actual error. When the houses cost on average $22.000, a 20% discrepancy on the value seems like something I'd want to keep in mind.