boston_housing

February 29, 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 [2]: # Importing a few necessary libraries
        import numpy as np
        import matplotlib.pyplot as pl
        import seaborn as sns
        import seaborn as sns
        from sklearn import datasets
        from sklearn.tree import DecisionTreeRegressor
        import pandas as pd

        # Make matplotlib show our plots inline (nicely formatted in the notebook)
        %matplotlib inline
        sns.set_palette("deep", desat = 0.6)
        sns.set_context(rc={"figure.figsize": (8,4)})
        sns.set_style("darkgrid")

        # 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!
```

```
In [3]: # insert data into a pandas dataframe to handle it easier
        df_housing_prices = pd.DataFrame(housing_prices)
        df_housing_features = pd.DataFrame(housing_features)
        l_features = ["CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS","RAD",
                      "TAX","PTRATIO","B","LSTAT"]
        l_descriptions=['CRIM: Per capita crime rate by town',
                        'ZN: Proportion of residential land zoned',
                        'INDUS: Proportion of non-retail business acres per town',
                        'CHAS: If tract bounds Charles River',
                        'NOX: Nitric oxides concentration',
                        'RM: Average number of rooms per dwelling',
                        'AGE: Proportion of owner-occupied units prior to 1940',
                        'DIS: Weighted distances to five Boston employment centres',
                        'RAD: Index of accessibility to radial highways',
                        'TAX: Full-value property-tax rate',
                        'PTRATIO: Pupil-teacher ratio by town',
                        'B: Proportion of afro-americans by town',
                        'LSTAT: % lower status of the population',
                      ]
        df_housing_features.columns = l_features
        df_housing_features.head()
```

```
Out[3]:       CRIM  ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD  TAX  PTRATIO  \
        0  0.00632  18   2.31     0  0.538  6.575  65.2  4.0900    1  296     15.3
        1  0.02731   0   7.07     0  0.469  6.421  78.9  4.9671    2  242     17.8
        2  0.02729   0   7.07     0  0.469  7.185  61.1  4.9671    2  242     17.8
        3  0.03237   0   2.18     0  0.458  6.998  45.8  6.0622    3  222     18.7
        4  0.06905   0   2.18     0  0.458  7.147  54.2  6.0622    3  222     18.7

                B  LSTAT
        0  396.90   4.98
        1  396.90   9.14
        2  392.83   4.03
        3  394.63   2.94
        4  396.90   5.33
```

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 [10]: # Number of houses in the dataset
         total_houses = df_housing_features.shape[0]

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

         # Minimum housing value in the dataset
         minimum_price = df_housing_prices.min().values[0]

         # Maximum housing value in the dataset
         maximum_price = df_housing_prices.max().values[0]

         # Mean house value of the dataset
         mean_price = df_housing_prices.mean().values[0]

         # Median house value of the dataset
         median_price = df_housing_prices.median().values[0]

         # Standard deviation of housing values of the dataset
         std_dev = df_housing_prices.std(ddof=0).values[0]

         # 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.

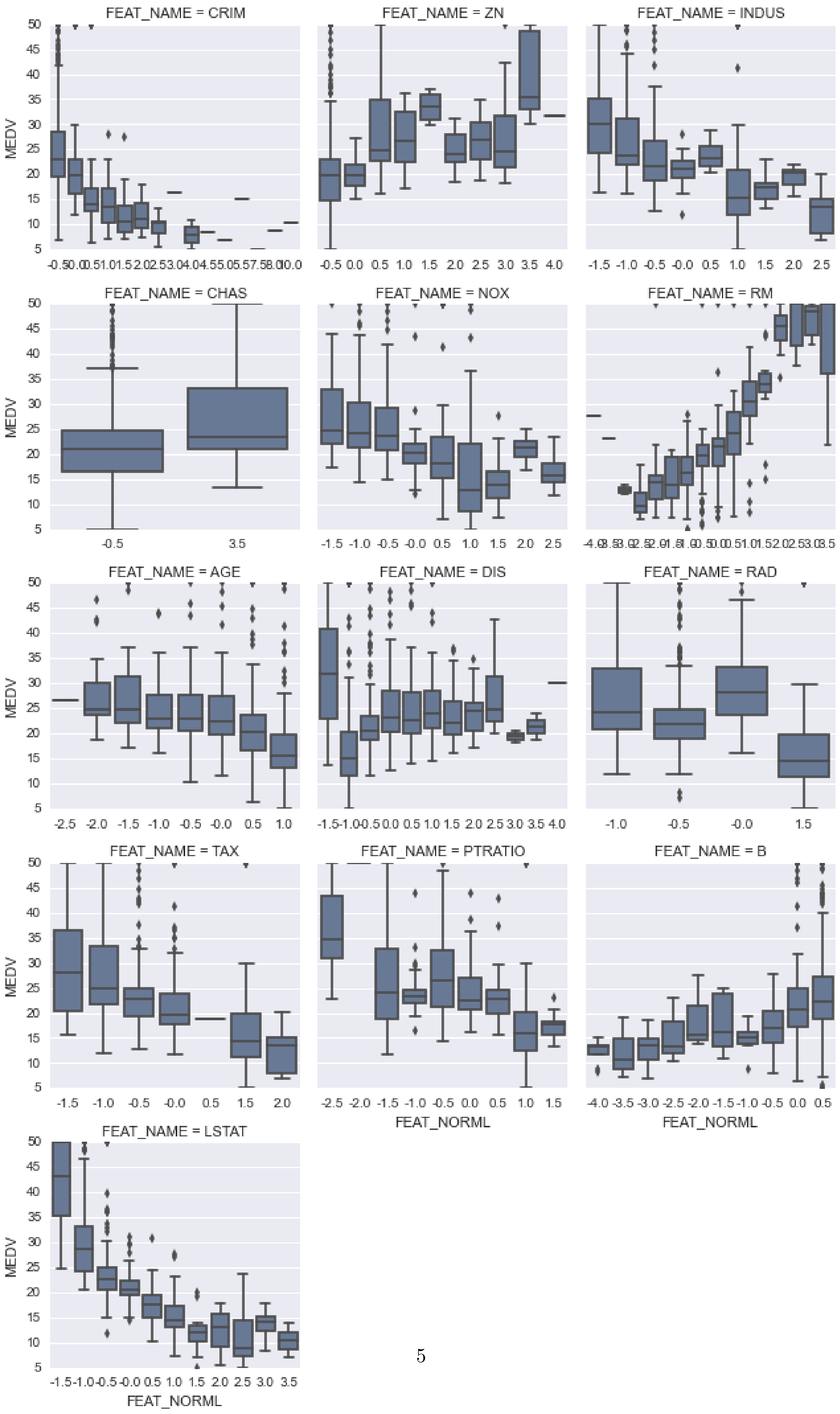
Remember, you can **double click the text box below** to add your answer!

Answer: To help me figure out what characteristics can be more adequate to use to predict the house prices, I am going to plot a boxplot for each feature related to the house prices in the dataset. For that, I've created the function `inplitData` to reshape data to be used with Seaborn's function `FacetGrid`. The Axis-axes below are in standard deviations to the mean of each feature.

```
In [4]: # A python library with some codes to help explore the dataset
        import additionalCode as addCd; reload(addCd);
        df_plot = addCd.inplitData(df_housing_features, df_housing_prices)
```

```
In [5]: g = sns.FacetGrid(df_plot, col="FEAT_NAME", col_wrap=3, sharex=False)
        g.map(sns.boxplot, "FEAT_NORML", "MEDV")
        # g.map(pl.scatter, "FEAT_VAL", "MEDV", alpha=.7)
```

```
Out[5]: <seaborn.axisgrid.FacetGrid at 0x103e7fc50>
```


FEAT_NAME = CRIM
FEAT_NAME = ZN
FEAT_NAME = INDUS
FEAT_NAME = CHAS
FEAT_NAME = NOX
FEAT_NAME = RM
FEAT_NAME = AGE
FEAT_NAME = DIS
FEAT_NAME = RAD
FEAT_NAME = TAX
FEAT_NAME = PTRATIO
FEAT_NAME = B
FEAT_NAME = LSTAT
MEDV
FEAT_NORML

As can be seen above, almost all features presented some kind of structure when related to the house prices, although I believe that the relationship of some of them would be less useful for this task. For instance, the proportion of residential land zoned (ZN) apparently is more informative just when its value is very low or very high. On all other values its relationship with prices is not clear. The behaviour for the weighted distance to Boston employment centers (DIS) also is mixed.

So, from my point of view, the most well-behaved features, that at least presented a "moving median" across all buckets and didn't show too much variation inside each bucket, were the following: - **LSTAT**: % lower status of the population - **RM**: Average number of rooms per dwelling - **TAX**: Full-value property-tax rate

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.

Answer: The values are:

```
In [6]: df_featClient = pd.DataFrame(CLIENT_FEATURES, columns=l_features).T
        l_client_featuresUsed = [[x[1][0] for x in df_featClient.ix[["LSTAT", "RM", "TAX"]].T.iteritems
        pd.DataFrame(CLIENT_FEATURES, columns=l_descriptions).T[0][[-1, 5, -4]]
```

```
Out[6]: LSTAT: % lower status of the population      12.130
        RM: Average number of rooms per dwelling      5.609
        TAX: Full-value property-tax rate           680.000
        Name: 0, dtype: float64
```

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 [7]: # 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, test_size=0.30,
                                                                 random_state=0)
            # Return the training and testing data subsets
```

```
    return X_train, y_train, X_test, y_test


# Test shuffle_split_data
try:
    X_train, y_train, X_test, y_test = shuffle_split_data(housing_features, 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: As pointed here, fitting a model and testing it on the same data would just repeat the labels of the sample. However, the goal of machine learning is to generalize from its experience and not just repeat what the algorithm has just seen. So, we usually split the data to check how the algorithm performs on yet-unseen examples. Also, holding out part of the data helps to figure out if the model is overfitted by comparing the error in the training and testing subsets. For instance, if the error in the training data is small and the error in the testing data start to climb at some point, it might indicate that the model is overfitted.

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 [8]: # 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:
            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: The first three metrics are related to classification task as they use counts from confusion matrix (like True Positives, False Negatives and so on). The fourth also is related to classification, but it is a weighted average of the precision and recall. The last metric, Mean Absolute Error, as pointed here, equally weighted all values when computing the average. So, I believe that the most appropriate metric for predicting housing prices is the Mean Squared Error that, besides other desirable charcteristics (as be differentiable), also gives more importance to outliers.

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 [9]: # Put any import statements you need for this code block
        from sklearn.metrics import make_scorer
        from sklearn.grid_search import GridSearchCV
        from pprint import pprint

        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
            # ========include necessary parameters =======
            scoring_function = make_scorer(performance_metric,
                                           greater_is_better=False)

            # Make the GridSearchCV object
            reg = GridSearchCV(regressor,
                               param_grid=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
            return reg


        # Test fit_model
        try:
            reg = fit_model(X_train, y_train)
            print "Successfully fit a model!"
        #     print "\nSet of parameters for best estimator:"
        #     pprint(reg.get_params())
        #     print "\nBest parameters from parameter grid:"
        #     pprint(reg.best_params_)
        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 algorithm searches through a parameter space for the set of parameters that yields the best result, as pointed here. This parameter space is a grid of all combinations from a list of possible values to each parameter that is passed to the algorithm. So, if you have a large number of parameters to test, grid search helps to test all combinations possible at once and find the best one.

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: Cross-validation procedure is a method to validate a model. Basically, it separates a sample into two or more subsets and uses one partition to train the model and another to verify its performance. Then, the procedure is run multiple times using different subsets to train and validate the model. The results are averaged over the rounds.

As mentioned before, it is a common practice to hold part of available data to be used as a test set to avoid overfitting. However, if we evaluate different settings of parameters over the same test data (and it is why we use cross-validation with grid search), we end up just tuning the parameters to a particular dataset, and not testing its generalization performance.

To overcame that, we could hold out yet another part of the dataset (also called validation set), use it to evaluate different parameters and then check its performance using the test set. Cross validation is used here. Instead of holding out a validation set and reduce even more the data available to train the model, we split the training set into different partitions and use one part to train and another to validate at each round.

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 [10]: 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):

                    s = int(s)
                    # 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 [11]: 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))
```

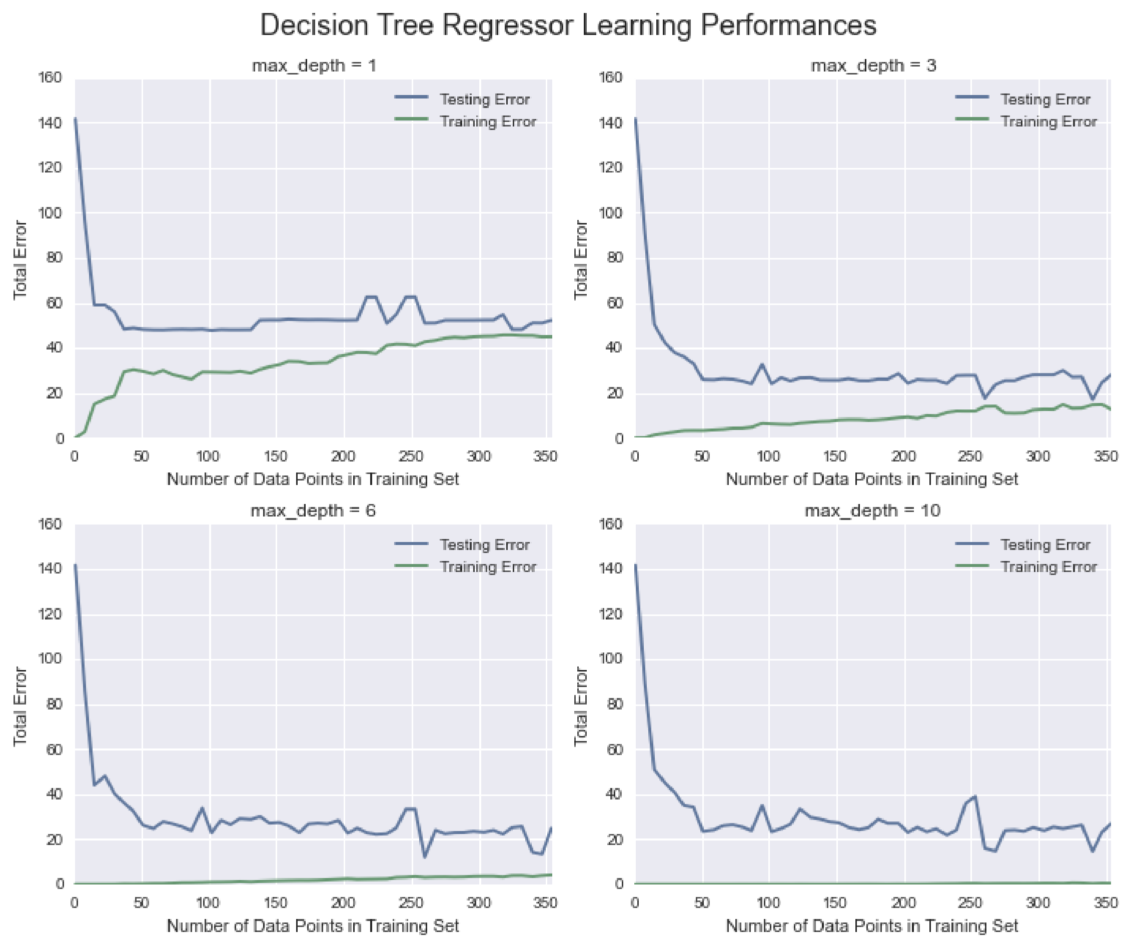
```
    test_err = np.zeros(len(max_depth))

    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()
```

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

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



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: Looking at the chart about the `max_depth=3`, it clearly shows that the error on the test data have already reached a Plato at the beginning (around 50 data points), but the training error has started to climb as the size of the training set was growing.

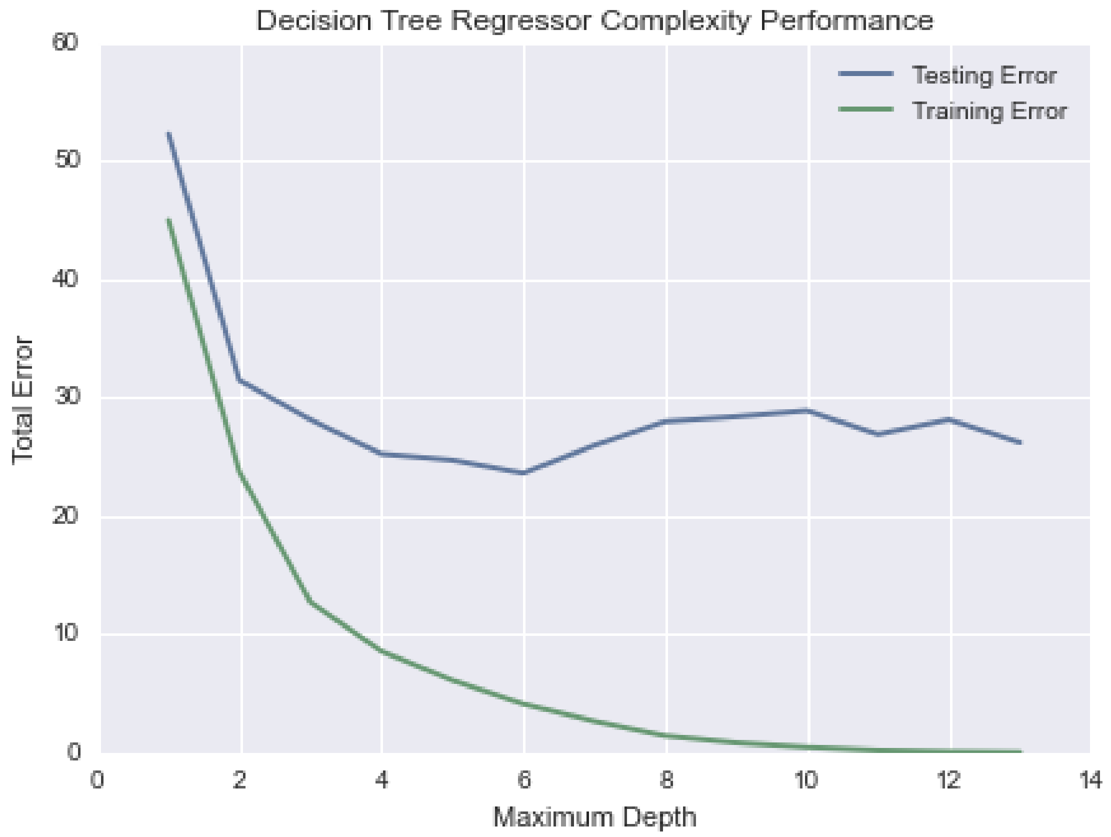
6.2 Question 8

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: Quoting Katie, from Udacity's "Intro to Machine Learning" course, High Bias happens when the model pays little attention to the data and High Variance occurs when it pays too much attention to the training set. So, in the `max_depth=1` graph, the model suffers from high bias, because it apparently is not sophisticated enough to capture the relationship between the features selected and the house prices. When the max depth is 10, the training error is the smallest of all models tested. However, its test error looks slightly greater than the error in the `max_depth=6` model. It can indicate that the model is overfitting the data and would not generalize well to new examples (high variance).

```
In [13]: 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: In the beginning, both training and test error became smaller when the maximum Depth was increasing. However, while the model complexity rose further, the training error kept going down, while the test error was increasing. In other words, in this case, when the complexity of the model increased, the variance became the primary concern while bias steadily fell.

As noticed in the last question, the maximum depth 10 had presented more Variance than the 6, which in turns apparently demonstrated less Bias than the maximum depth 4. So, I would say that the parameter that would best generalize to new instances would be the Maximum Depth 6.

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, 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 [14]: def iterateGridSearch(X, y):
             """
             Run multiples iterations to find the maximum depth fro the tree. Return a
             dictionary with a model with each depth found and a list of the best
             parameter obtained at each iteration
             :param X_train: numpy array. The features used to train the model
             :param y_train: numpy array. The output desired of each instance
             """
             l_OptimalMaxDepth = []
             d_rtn = {}
             for i_try in xrange(1, 500):
                 reg = fit_model(X, y)
                 l_OptimalMaxDepth.append(reg.best_params_)
                 if reg.best_params_ not in d_rtn.keys():
                     d_rtn[reg.best_params_["max_depth"]] = reg

             return l_OptimalMaxDepth, d_rtn
```

```
In [15]: # plot the distribution of the best parameters found
         l_OptimalMaxDepth, d_rtn = iterateGridSearch(housing_features, housing_prices)
         df_plot = pd.DataFrame(l_OptimalMaxDepth)
         df_plot["count"]=None
         df_plot["count"]=1
         df_plot.groupby("max_depth").count()
```

```
Out[15]:            count
         max_depth
         4            259
         5             14
         6            114
         7             61
         8             29
         9             12
         10            10
```

```
In [16]: print "median of the iterations: {}".format(df_plot.median().values[0])
         print "mean of the iterations: {:0.2f}".format(df_plot.mean().values[0])
```

```
median of the iterations: 4.0
mean of the iterations: 5.32
```

```
In [17]: reg = d_rtn[4]
         print "Final model optimal parameters:", reg.best_params_
```

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

Answer: Above I summarized 500 iterations made to see which was the best parameter found at each one. The mode and the median of all test were 4, despite that `max_depth=6` also presented a high counting. I am going to use the `max_depth=4` to predict the house price.

7.2 Question 11

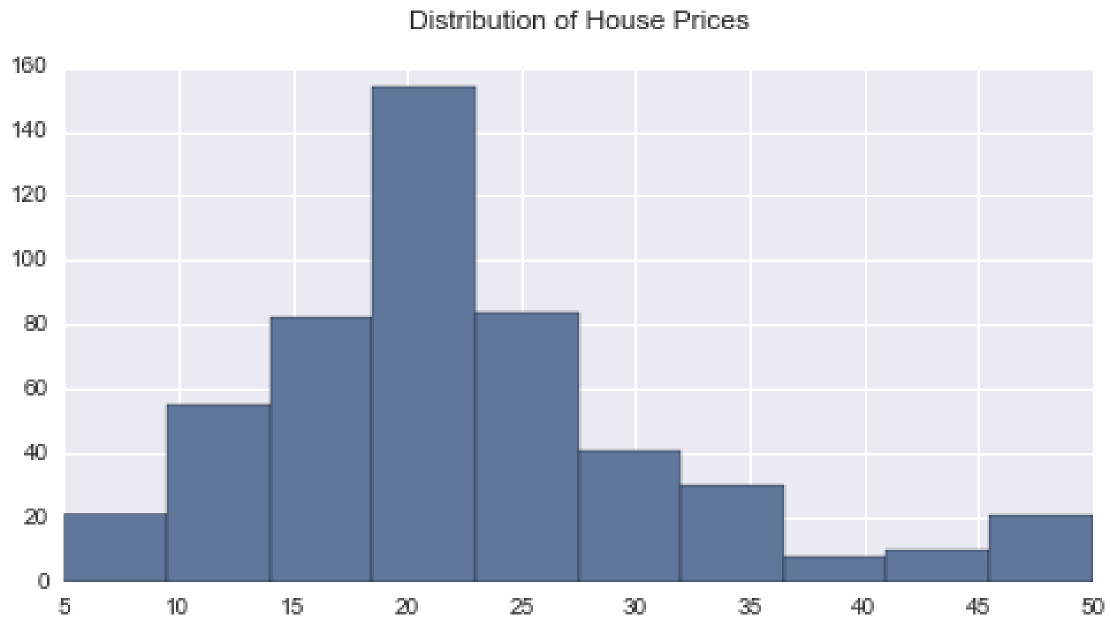
With your parameter-tuned model, what is the best selling price for your client's home? How does this selling price compare to the basic statistics you calculated on the dataset?

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 [130]: l_ax = pd.DataFrame(list(y_test)+list(y_train)).hist(bins = 10)
          l_ax[0][0].set_title("Distribution of House Prices\n");
```



Answer: The regressor predicted value to the client's home of 21.630, which is less than a standard deviation (9.2) from the mean of house prices (21.2), as can be checked in the histogram above.

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: Although this model is a good start, it might not be the more efficient. Another benefit of reducing dimensions, as pointed out by Charles and Michael, from Georgia Tech, when they talked about the curse of dimensionality, is that fewer data would be needed to build a model that accurately generalize to unseen examples.

Style notebook

```
In [11]: #loading style sheet
         from IPython.core.display import HTML
         HTML( open('ipython_style.css').read())
```

```
Out[11]: <IPython.core.display.HTML object>
```