Project 1: Boston Housing Prices

Patrick Martin

This project required synthesis of introductory modeling and evaluation material. Using the provided code skeleton, I modified four sections of the code to collect basic statistics from provided data, implement a performance metric, analyze the learning curves and model complexity, and tune a `DecisionTreeRegressor` algorithm.

Statistical Analysis of Housing Data

The Boston housing data contained **506** entries that each have **13** features. Using `numpy` array functions, I determined the following statistical measures:

Minimum home price = **5.0**

Maximum home price = **50.0**

Mean home price = **22.5328**

Median home price = **21.2**

Standard deviation = **9.188**

Model Evaluation

Home prices are based on continuous numerical values (or, drawn from $\mathbb{R}$). Consequently, predicting a new home price based on current home price data is a **regression** problem. Choosing an error-based metric fits the problem, since we want the prediction algorithm to place values as close as possible to comparable homes. In particular, I selected the **mean-squared error** metric to penalize incorrect predictions far more than an absolute error. The other scoring mechanisms discussed in class (e.g. precision, recall) make most sense for classification problems, not regression.

When tuning the `DecisionTreeRegressor`, we must split the data into appropriately sized training and testing data, which in this case was split 70/30. Performing this split was enabled by a simple function call to `train_test_split`. We need to split the data into randomly chosen bins of training/test sets to get the best performance possible. If you do not do the split correctly, it will result in a learning algorithm with poor learning performance (too little training data) or poor validation (too little test data). In general, machine learning algorithms need cross validation to tune the algorithm using more sub sets of data from the given set. For example, K-fold cross validation will run K experiments with a randomly chosen set of data. It will also be validated K times.

The cross validation tool we used in this project was `GridSeachCV` that accepts a candidate algorithm, the parameters of interest, and a scoring function. By leveraging a known algorithm, the best `max_depth` of the `DecisionTreeRegressor` is computed with an exhaustive search. My particular implementation uses the same `mean_squared_error` performance metric and passes in the `max_depth` parameter array.

Model Performance

The provided skeleton code created a loop to increase the maximum decision tree depth size and plot the error results. Figures 1, 2, 3 illustrate these curves for different `max_depth` values. As the depth of the tree increases, the overall error for training and test data inputs drops with the size of the training set.

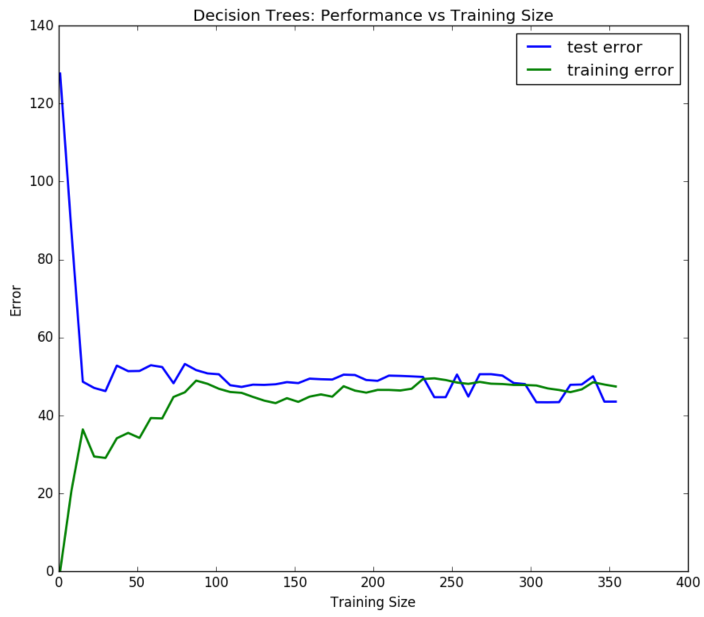

Figure 1: A plot of the training and test data error over size of training. This image is the initial run with the decision tree `depth = 1`.

Figure 1 shows an interesting case where there is considerable bias with `max_depth=1`. When the depth increases to 4, the training error drops significantly as the training data size increases. The test error also drops, but does not converge to the training error curve. The mean squared error hovers close to 20 for training sets over 200 data points, which seems to These plots indicate that the algorithm is fully trained with at least 200 data points and a max depth of at least 4. This model still has some variance, due to the difference between training data error and test data error. One interesting trend is that the plots based on depths 5 through 10 do not increase the test error significantly. This observation indicates that higher depth does not always mean better performance for the added computational cost.

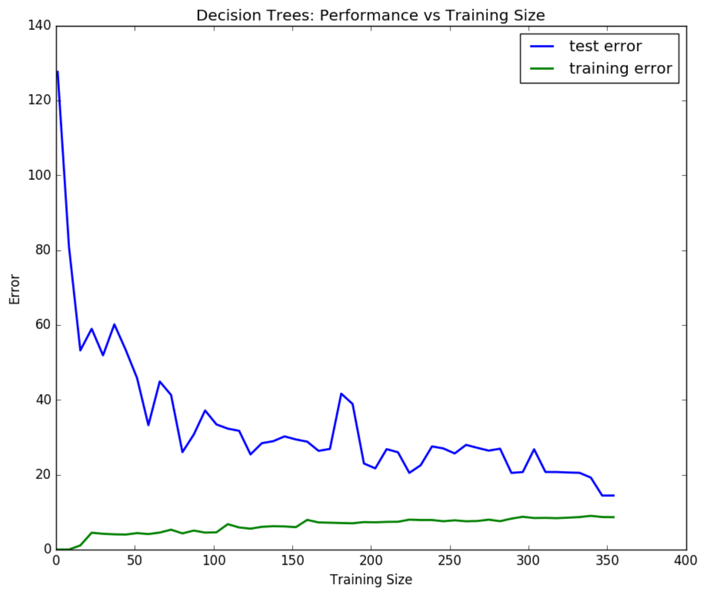

Figure 2: This image shows the error with a decision tree `depth = 4`.

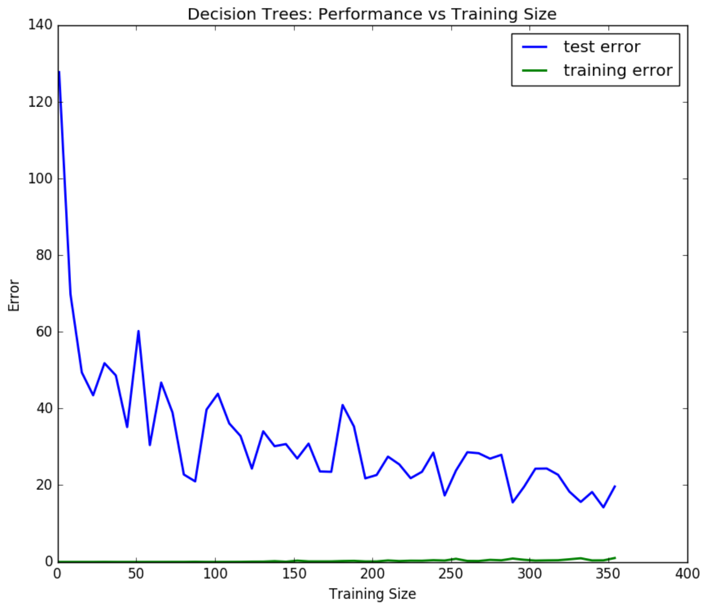

Figure 3: This image shows the error with a decision tree `depth = 10`.

By plotting the test error against the depth of the the `DecisionTreeRegressor`, shown in Figure 4, I get confirmation of the trend seen in the learning curves. Although **training error approaches 0**, the test error continues to **remain between 10 and 20**. Given this plot, I would recommend a `DecisionTreeRegressor` model with `max_depth` $\in \{4, 5, 6\}$.

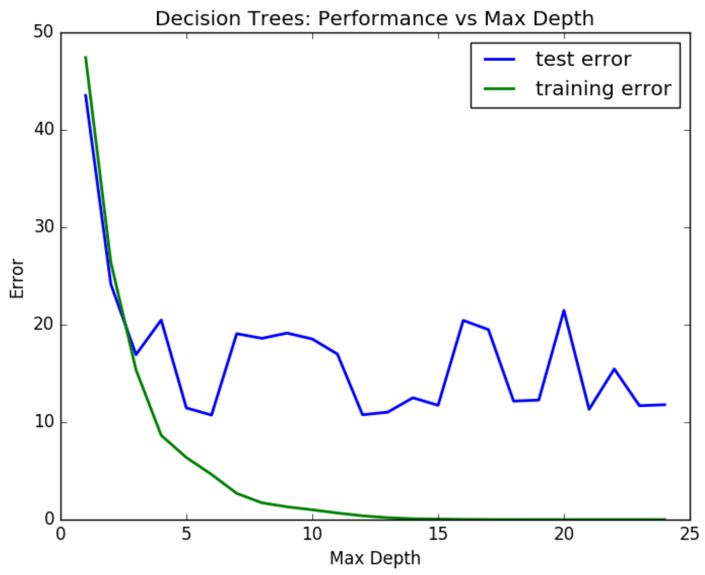

Figure 4: A graph of the model complexity as the depth of the decision tree increases.

Model Prediction

Using the `GridSearchCV` discussed before, I ran the program several times. On the whole, the tuned model resulting from this grid search based on mean-squared error had a `mzx_depth=4`, with the occasional value of `5` or `6`. One sample run of the main program iterated the model fit function five times and I saw the following output:

```
                    depth = [4, 6, 9, 4, 4]
      predictions = [21.6297, 20.7659, 19.3272, 21.6297, 21.6297]
```

These results confirm that the grid search tunes to model to favor a **tree depth of 4**. Based on the consistent prediction values, I think this model is a reasonable predictor of housing prices.