UDACITY

MACHINE LEARNING – NANODEGREE

BOSTON HOUSING PRICES

BY: TODD FARR

1 Statistical Analysis and Data Exploration

1.1 Statistical Analysis

The calculated statistics for the data set using the Python Numpy library are as follows:

- The total number of houses in the Boston Housing data set is: 506
- The total number of features associated with each house is: 13
- The minimum housing price is: $5,000.00
- The maximum housing price is: $50,000.00
- The mean housing price is: $22,532.81
- The median housing prices is: $21,200.00
- The Standard Deviation is: 9.188

1.2 Data Exploration

To gain a better understanding of how the features are related to the target housing price, each feature was plotted vs the price. This method can provide quick visual insight into possible correlations between the target and certain features. It can also be a method to help identify what features carry the most weight or are the most important when trying to develop a model that best generalizes the data (Figure 1).

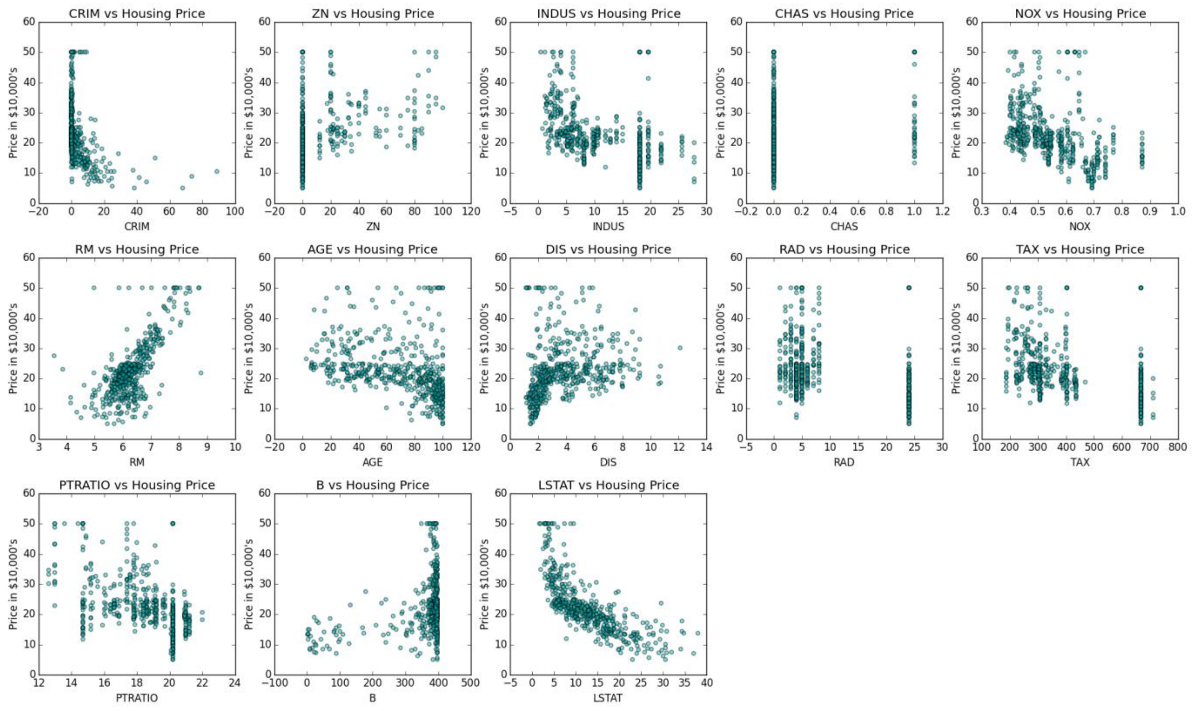

Figure 1: Each dataset feature plotted against the corresponding house price.

2 Evaluating Model Performance

2.1 Model Error Metrics

The Scikit-Learn Library offers three different types of error metrics for evaluating regression model performance: *Median Absolute Error (MedAE), Mean Absolute Error (MAE) and Mean Squared Error (MSE).*

The *Median Absolute Error (MedAE)* is calculated by taking the median of all the absolute differences between the target and predicted value as depicted by Equation 1.

$$MedAE(y, \hat{y}) = median(|y_1 - \hat{y}_1|, \ldots, |y_n - \hat{y}_n|) \tag{1}$$

Because *MedAE* is the only provided error metric that relies on the median and not the average of a series of error values it is robust to the effects of outliers. However, during the data exploration and experimentation phase it was apparent that this dataset is relatively evenly distributed and the model would not benefit from this specific characteristic.

The *Mean Absolute Error (MAE)* calculates the average of all the relative errors between the target and predicted values as depicted by Equation 2.

$$MAE(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} |\, y_i - \hat{y}_i| \tag{2}$$

It's important to note that a side effect of taking the absolute of the relative distances is that negative and positive errors do not cancel each other out when calculating this value. During experimentation and in theory this metric seems to behaves similarly to *MSE* on this particular dataset.

The *Mean Squared Error (MSE)* calculates the average of all the relative errors squared between the target and predicted values as depicted by Equation 3.

$$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} (\, y_i - \hat{y}_i)^2 \tag{3}$$

Like *MAE*, squaring the relative distances also ensures that positive and negative error values do not cancel each other, but this equation also exhibits other unique characteristics: it penalizes large error values more so than those that are small and the function is differentiable. The final decision to choose this metric also stems from the use of the Scikit-Learn Library Decision Tree Regressor algorithm, in which the documentation states that *MSE* is the only supported criterion for determining the quality of a split.

2.2 Splitting the Data

It's important to partition data into a training set and testing set as this allows us to evaluate the performance of the model based on this independent data and then use these performance metrics to prevent overfitting. The data points from our test set with a known classification or value can be compared to the classification or value of that predicted by the model which was trained via the training set. From these two classifications or values the associated error can be calculated, which allows for the creation of a closed loop feedback system for the model.

If the dataset was not partitioned, the model would have already seen all the available data points and would simply regurgitate the labels or values associated with each data point. This would result in a model appearing to be a perfect representation of the dataset (error ~0). However, this function has the tendency to be highly-complex in order to best fit every data point, meaning it will not generalize the data well and will not provide any useful output on unseen data points (Figure 2).

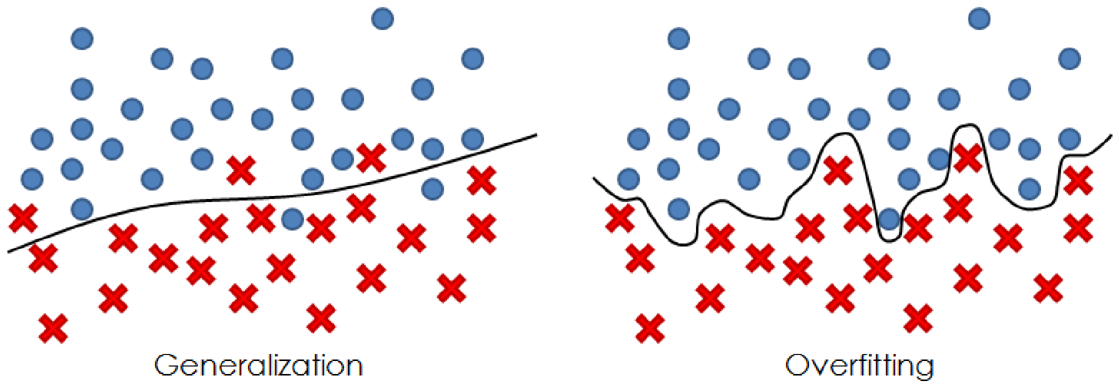

Figure 2: A graphical representation of a generalization vs overfitting

2.3 Grid Search

Grid Search is an algorithm that iterates through a supplied model and parameter dictionary and attempts to find the optimal parameters from this dictionary that returns the best cross-validation estimator performance score from the supplied metric. In this particular instance Grid Search is trying to find the optimal Max Depth value for the Decision Tree Regression algorithm using the *MSE* metric. Applying the *best_params_* method to the Grid Search Object after fitting reveals that for this model the optimal Max Depth value is 4 or 5 depending on the randomness introduced by the code. This value can then be visually confirmed by examining the Model complexity plot (Figure 3).

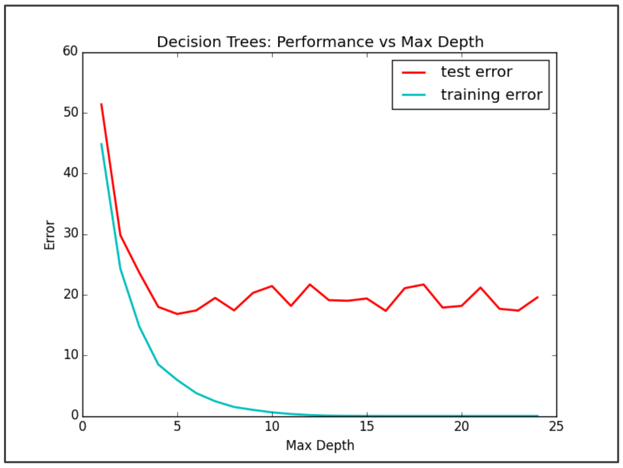

Figure 3: The correlation between max depth and test error vs training error

In this graph the test error value levels off and actually increases after an approximate Max Depth equal to 4. This indicates the model is not generalizing the data any better, and in some cases worse, as the Max Depth is increased beyond this optimal value.

2.4 Cross-Validation

Cross-Validation is a common technique used to prevent overfitting. The basic principle behind this classification of model evaluation methodology is by dividing the data into training and testing data the model can be exposed to a dataset that so far has been unseen by it. Since the actual target values for the test set are known they can be compared to the predicted values to compute an error metric for that particular data point. If this is done enough times a metric can be computed from these accumulated errors and provide the closed loop feedback system necessary for optimizing the model. This generalization encompasses the simplest form of cross-validation and aligns with the principles used in the splitting of the data section above. This is often referred to as the holdout method.

A common problem with the holdout method is that results can vary drastically depending on where the data division is made. A remedy to this problem, and the default technique used by the Grid Search CV algorithm, is the k-fold cross-validation method. This is an iterative process in which the dataset is divided into *k* partitions, and then processed through the aforementioned holdout method *k* times. Each iteration results in one of the *k* subsets assuming the role of the test set as the other *k*-1 partitions are combined into a training set. The total error is then calculated by averaging the errors from each iteration. A graphical representation of this cross-validation method can be seen in Figure 4.

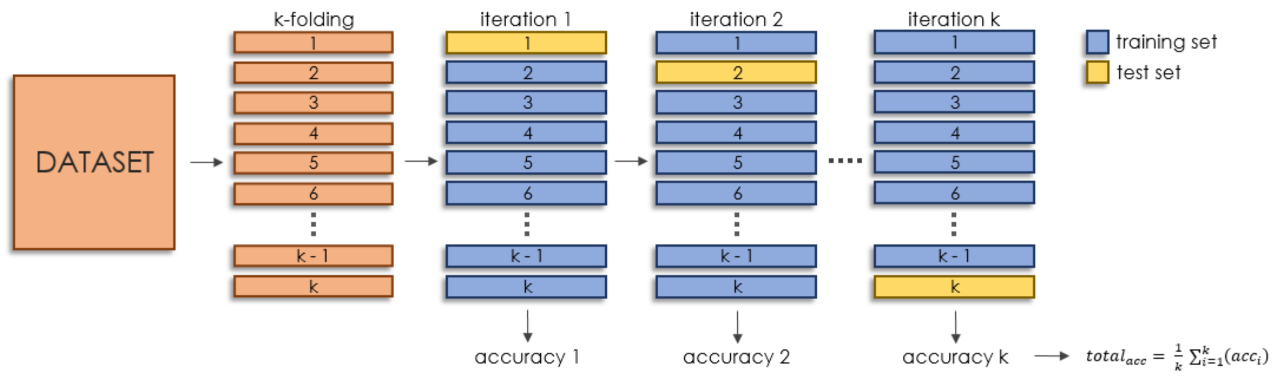

Figure 4: k-fold cross-validation visualized

The k-fold cross-validation method can be particularly useful if the dataset is limited as it functions on the creation of *k* distinct training sets which allows for the use of all data points for validation exactly once. The disadvantage, as like with any iterative process, is that it has to start from scratch *k* times. This makes choosing a value for *k* an important task as a high *k* value, while less biased, can suffer from high variance and be computationally expensive. However the flip side of this argument is that if the *k* value is too low it may have the tendency to be bias depending on the models performance in relation to the training set size.

Technically, because this model does not have a substantial slope at the training size created by a *k*-fold cross-validation value of 3 (~337 data points) this default value of the Grid Search CV algorithm is okay to use (Figure 5). However, one could argue that the more widely accepted standard of 5-fold or 10-fold cross-validation could also be applied in this case.

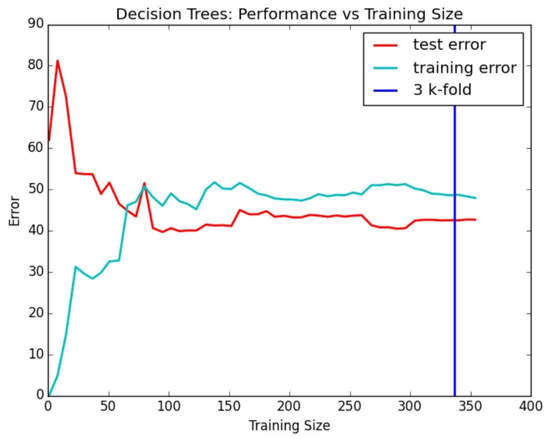

Figure 5: The 3 k-fold cross validation training size crosses at a stable portion on the training error curve

3 Analyzing Model Performance

3.1 The Effects of Training Size

As the training size increases the test and training errors have opposing effects (Figure 6). While it's true they both begin to stabilize, the general trend is that the training error is increasing while the test error is decreasing. This makes perfect sense because as the machine learning algorithm is exposed to more data it can better generalize it as opposed to trying to distinguish a classification or fit from only a limited number of data points.

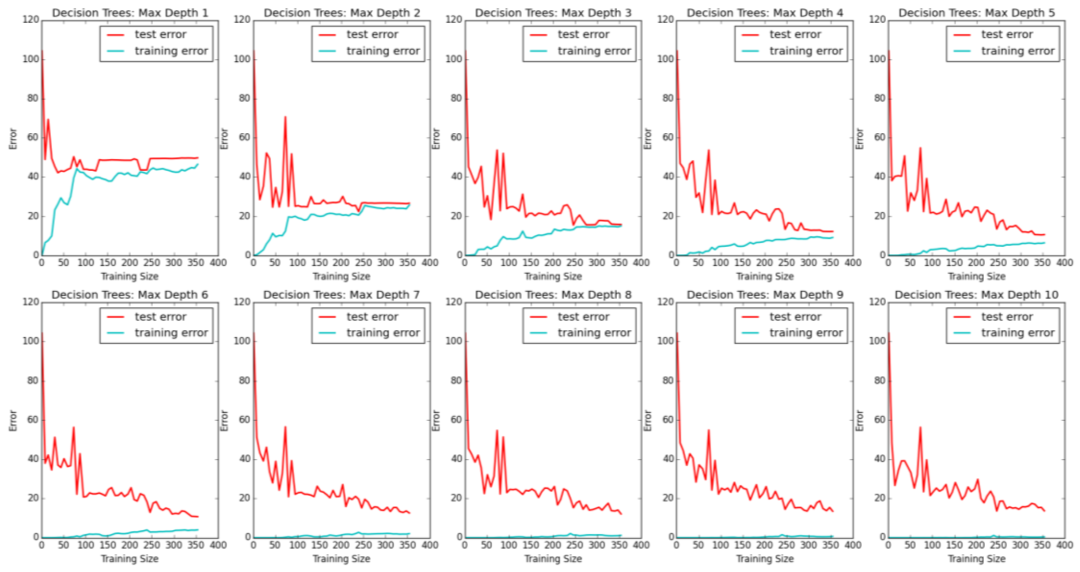

Figure 6: The Learning Curves from Max Depth 1-1

3.2 A Fully Trained Model

With a Max Depth = 1 the model suffers from underfitting or can be classified as having high bias. This can be visually verified as the model exhibits high training error even as the training size is increased indicating that it has not captured the underlying trend yet. However, when the model is fully trained (i.e. Max Depth = 10) it suffers from overfitting and can be classified as being Highly Variant. This can be clearly seen by how well the model can predict the training data (error ~0) but still has trouble predicting for the data points from the test set (Figure 6).

3.3 Model Complexity

As the model complexity (Max Depth) increases the error on the training set converges to be approximately zero, whereas the error on the test set stabilizes and even increases after a certain level of complexity. This indicates the model is overfitting the data after this point. Depending on how the data is split for the test and training sets this stabilization occurs somewhere between a model complexity with Max Depth 4-6 with a majority of the cases being equal to 4. This indicates that the optimal model complexity has a MD value approximately equal to this value (Figure 7).

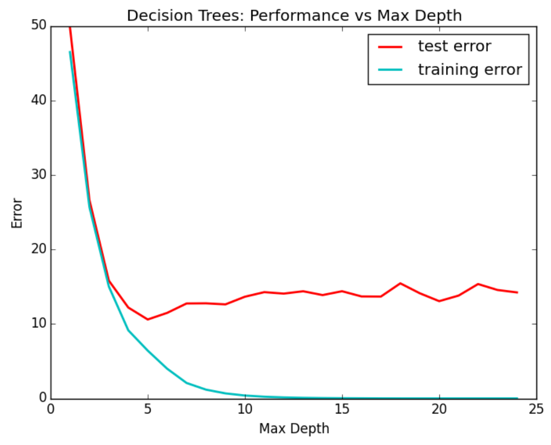

Figure 7: Test and Training Error vs Model Complexity

4 Model Prediction

4.1 The Final Model

Since the Grid Search object was only supplied with the Max Depth parameter for the Decision Tree Regressor, only this parameter was optimized and returned by the class. The final Model object can be seen below:

```
GridSearchCV(cv=None, error_score='raise',
       estimator=DecisionTreeRegressor(criterion='mse', max_depth=None,
              max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
              min_samples_split=2, min_weight_fraction_leaf=0.0, random_state=None,
              splitter='best'),
       fit_params={}, iid=True, loss_func=None, n_jobs=1,
       param_grid={'max_depth': (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)},
       pre_dispatch='2*n_jobs', refit=True, score_func=None,
       scoring=make_scorer(mean_squared_error, greater_is_better=False), verbose=0)
```

Depending on how the data set was split the optimal max depth was between 4 and 6. However in most cases the Grid Search returned a Max Depth of 4 from its _*best_params* method. Using this max depth the final predicted output for the house price can be seen in Figure 8.

Figure 8: The Model's final predicted housing price

4.2 Comparing Prediction

The predicted value of $21,629.74 does not raise immediate red flags as it's fairly close to both the mean and the median house prices for the dataset. Taking the verification of this a step a little further, the data was loaded into a Pandas DataFrame which was then filtered for all housing values $21,000.00 <= x <= $22,000.00. From previous data exploration it appeared there was a correlation between the house price and the following two features: RM - the average number of rooms per dwelling and LSAT - % lower status of the population. Utilizing the filtered DataFrame the mean for these values was then compared to the prediction values and this again appears to support the prediction data point (Table 1).

	DataFrame Mean Value	Prediction Data Point
RM Feature	6.20	5.61
LSAT Feature	11.52	12.13

Table 1: Mean of filtered data features vs prediction data point features

Additionally, aside from manually splitting the data in relation to the target price to compare the important features, the Scikit-Learn NearestNeighbors class was also implemented to identify and calculate a mean target housing price of $21,520.00 for the 10 nearest neighbors. This results in only a 0.5% difference when compared to the predicted value from the model.