# Project 1 - Predicting Boston Housing Prices - Veeresh Taranalli

January 2, 2016

## 1 Statistical Analysis and Data Exploration

We explore the Boston Housing dataset and answer the following questions about the dataset.

- Q. Number of data points (houses)?
  A. 506

- Q. Number of features?
  A. 13

- Q. Minimum and maximum housing prices?
  A. Minimum price: 5000.0 USD, Maximum price: 50000.0 USD

- Q. Mean and median Boston housing prices?
  A. Mean price: 22532.81 USD, Median price: 21200.0 USD

- Q. Standard deviation?
  A. Standard deviation of house prices: 9188.01 USD

## 2 Evaluating Model Performance

In this section, we answer the following questions regarding evaluating the prediction model performance.

- Q. Which measure of model performance is best to use for predicting Boston housing data and analyzing the errors? Why do you think this measurement most appropriate? Why might the other measurements not be appropriate here?
  A. As the Boston housing prices are continuous in nature (floating point values) and we are using a regression model, the mean squared error appears to be a good metric for model performance evaluation as the housing price distribution does not appear skewed or there aren't too many outliers. Other metrics available for regression models in scikit-learn are the median absolute error, mean absolute error, r2 and variance_explained. The median absolute error is not applicable as it only reports the median of the absolute errors and would be more useful if there were outliers in the dataset or the housing price distribution was skewed which is not the case here. The r2 and variance_explained could be used but these scores would be harder to interpret. The mean absolute error is very close to the mean squared error and could also be used.

- Q. Why is it important to split the Boston housing data into training and testing data? What happens if you do not do this?
  A. It is necessary to split the Boston housing data into training and testing data to obtain an accurate

estimate of the prediction model performance on unseen data i.e., to estimate how well our prediction model generalizes. If we do not split the dataset and use all the available data for training our model, we might end up overfitting the data (by choosing a more complex model) as the training set error always decreases as the model complexity increases.

- Q. What does grid search do and why might you want to use it?
  A. Grid Search is useful for fine tuning our prediction model based on a particular ML technique. In this project we are using a Decision Tree Regressor ML technique. Grid Search chooses the best model based on a scoring metric (mean squared error in our project) for all possible choices of the chosen model parameters.

- Q. Why is cross validation useful and why might we use it with grid search?
  A. The gridsearchCV function (in scikit-learn) we have used uses a K-fold cross validation approach, by default K = 3. K-fold cross validation of our prediction model is done by dividing the training data into K subsets of equal size where we obtain the performance measure of our model as the average scores obtained when we let each subset be the validation/test set and use the rest of the subsets for training our model. This process has two advantages. The cross validation ensures that the model parameter tuning does not overfit to any kind of pattern in the train/test split we performed and ensures that the model parameters are tuned to perform uniformly well on all of the training data available to us. The second advantage is that cross validation is useful when we have a limited amount of training data, but still want to get a good estimate of how well our prediction model would perform on the test set, without actually using the test set i.e., how well our model would generalize for unseen inputs.

# 3 Analyzing Model Performance

- Q. Look at all learning curve graphs provided. What is the general trend of training and testing error as training size increases?
  A. The training error increases and the testing error decreases as the training size increases.

- Q. Look at the learning curves for the decision tree regressor with max depth 1 and 10 (first and last learning curve graphs). When the model is fully trained does it suffer from either high bias/underfitting or high variance/overfitting?
  A. When the model is fully trained (i.e., using all the training data available), the decision tree regressor with max depth 1 suffers from high bias/underfitting. We reach this conclusion because both the training and testing errors close to one another and are quite large. Whereas, the decision tree regressor with max depth 10 suffers from high variance/overfitting. Again, from the learning curve, it is clear the training error is very small whereas the testing error is still quite large compared to the training error, which shows that the model is not able to generalize well to new data.

- Q. Look at the model complexity graph. How do the training and test error relate to increasing model complexity? Based on this relationship, which model (max depth) best generalizes the dataset and why?
  A. From the model complexity graph, the training error decreases as the model complexity (max depth) increases. The testing error also decreases with increasing model complexity upto a certain point and then it appears to slightly increase/remain same. Therefore, the model that best generalizes the dataset is when max depth = 4 as the testing error is the lowest. Note that this is also the final model chosen by the grid search.

# 4 Model Prediction

- Q. Model makes predicted housing price with detailed model parameters (max depth) reported using grid search. Note due to the small randomization of the code it is recommended to run the program several times to identify the most common/reasonable price/model complexity. Compare prediction to earlier statistics and make a case if you think it is a valid model.
  A. The final model chosen by the grid search is a model with max depth = 4 and this model predicts the housing price as 21.63k USD for the given input feature vector. From the dataset we find through inspection a feature vector that is very close to the given feature vector, specifically this [11.95110, 0.00, 18.100, 0, 0.6590, 5.6080, 100.00, 1.2852, 24, 666.0, 20.20, 332.09, 12.13] whose price is given as 27.90k USD. The mean squared testing error for the max depth = 4 model as seen from the learning curve graph is approx. 20 which translates to an error of about 4k USD for the housing price prediction. Based on the above observations, it appears that the decision tree regressor model is a valid/good model for predicting the housing prices.

# 5 Python Code for the Project

In [11]:
```
"""Load the Boston dataset and examine its target (label) distribution."""

%matplotlib inline
import warnings
warnings.filterwarnings('ignore')


# Load libraries
import numpy as np
import pylab as pl
from sklearn import datasets
from sklearn.tree import DecisionTreeRegressor

###################################
### ADD EXTRA LIBRARIES HERE ###
###################################

from sklearn import cross_validation, grid_search
from sklearn.metrics import mean_squared_error, make_scorer

def load_data():
    """Load the Boston dataset."""

    boston = datasets.load_boston()
    return boston

def explore_city_data(city_data):
    """Calculate the Boston housing statistics."""

    # Get the labels and features from the housing data
    housing_prices = city_data.target
    housing_features = city_data.data

    ###################################
    ### Step 1. YOUR CODE GOES HERE ###
    ###################################
```

```
    # Please calculate the following values using the Numpy library

    # Size of data (number of houses)?
    print "Size of data (number of houses):", housing_features.shape[0]

    # Number of features?
    print "Number of features:", housing_features.shape[1]

    # Minimum price?
    # Multiply by 1000 as house prices are in the units of £1000's
    print "Minimum price:", np.min(housing_prices)*1000, "USD"

    # Maximum price?
    # Multiply by 1000 as house prices are in the units of £1000's
    print "Maximum price:", np.max(housing_prices)*1000, "USD"

    # Calculate mean price?
    # Multiply by 1000 as house prices are in the units of £1000's
    print "Mean price:", np.mean(housing_prices)*1000, "USD"

    # Calculate median price?
    # Multiply by 1000 as house prices are in the units of £1000's
    print "Median price:", np.median(housing_prices)*1000, "USD"

    # Calculate standard deviation?
    # Multiply by 1000 as house prices are in the units of £1000's
    print "Standard deviation of house prices:", np.std(housing_prices)*1000, "USD"


def split_data(city_data):
    """Randomly shuffle the sample set. Divide it into 70 percent training and 30 percent test

    # Get the features and labels from the Boston housing data
    X, y = city_data.data, city_data.target

    ###################################
    ### Step 2. YOUR CODE GOES HERE ###
    ###################################

    # Use the train_test_split function sklearn.cross_validation
    # random_state = 131 does the random shuffling
    X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y,
                                                                         test_size=0.3,
                                                                         random_state=131)

    return X_train, y_train, X_test, y_test

def performance_metric(label, prediction):
    """Calculate and return the appropriate error performance metric."""

    ###################################
    ### Step 3. YOUR CODE GOES HERE ###
    ###################################
```

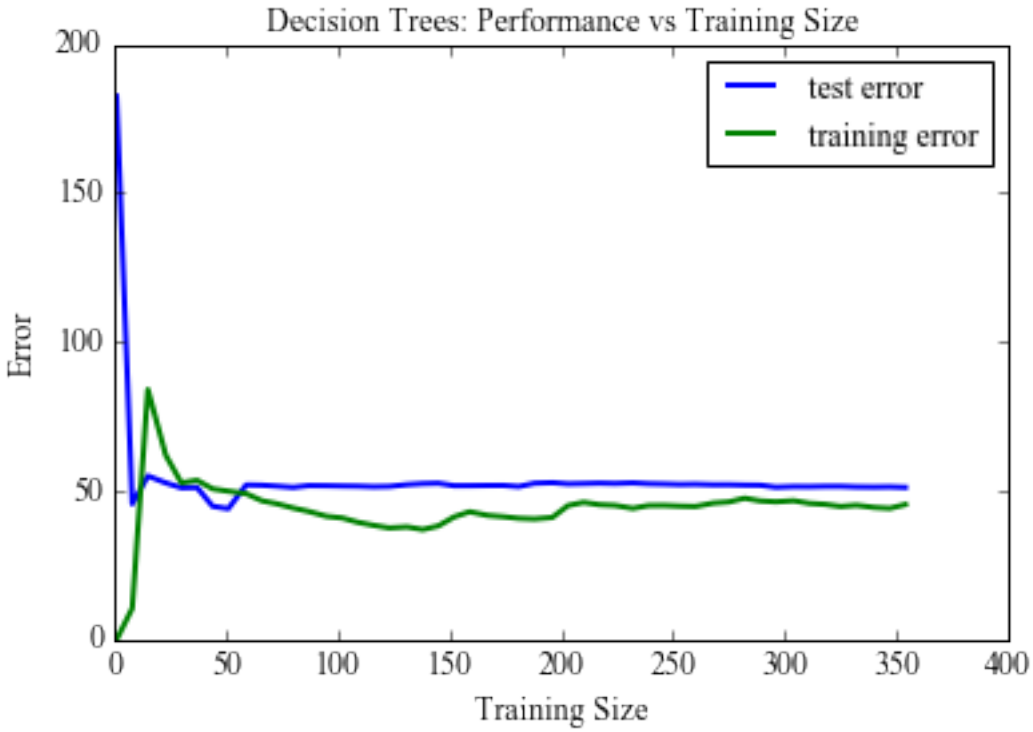
```
    # The following page has a table of scoring functions in sklearn:
    # http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics

    # As we are using decision tree regressor model for prediction,
    # we use the mean_squared_error performance metric.
    return mean_squared_error(label, prediction)


def learning_curve(depth, X_train, y_train, X_test, y_test):
    """Calculate the performance of the model after a set of training data."""

    # 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))

    print "Decision Tree with Max Depth: "
    print depth

    for i, s in enumerate(sizes):

        # Create and fit the decision tree regressor model
        regressor = DecisionTreeRegressor(max_depth=depth)
        regressor.fit(X_train[:s], y_train[:s])

        # Find the performance on the training and testing set
        train_err[i] = performance_metric(y_train[:s], regressor.predict(X_train[:s]))
        test_err[i] = performance_metric(y_test, regressor.predict(X_test))


    # Plot learning curve graph
    learning_curve_graph(sizes, train_err, test_err)


def learning_curve_graph(sizes, train_err, test_err):
    """Plot training and test error as a function of the training size."""

    pl.figure()
    pl.title('Decision Trees: Performance vs Training Size')
    pl.plot(sizes, test_err, lw=2, label = 'test error')
    pl.plot(sizes, train_err, lw=2, label = 'training error')
    pl.legend()
    pl.xlabel('Training Size')
    pl.ylabel('Error')
    pl.show()


def model_complexity(X_train, y_train, X_test, y_test):
    """Calculate the performance of the model as model complexity increases."""

    print "Model Complexity: "

    # We will vary the depth of decision trees from 2 to 25
    max_depth = np.arange(1, 25)
```

```
    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
    model_complexity_graph(max_depth, train_err, test_err)


def model_complexity_graph(max_depth, train_err, test_err):
    """Plot training and test error as a function of
    the depth of the decision tree learn."""

    pl.figure()
    pl.title('Decision Trees: Performance vs Max Depth')
    pl.plot(max_depth, test_err, lw=2, label = 'test error')
    pl.plot(max_depth, train_err, lw=2, label = 'training error')
    pl.legend()
    pl.xlabel('Max Depth')
    pl.ylabel('Error')
    pl.show()


def fit_predict_model(city_data):
    """Find and tune the optimal model.
    Make a prediction on housing data."""

    # Get the features and labels from the Boston housing data
    X, y = city_data.data, city_data.target

    # Setup a Decision Tree Regressor
    regressor = DecisionTreeRegressor()

    parameters = {'max_depth':(1,2,3,4,5,6,7,8,9,10)}

    ####################################
    ### Step 4. YOUR CODE GOES HERE ###
    ####################################

    # 1. Find an appropriate performance metric. This should be the same as the
    # one used in your performance_metric procedure above:
    # http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html
```

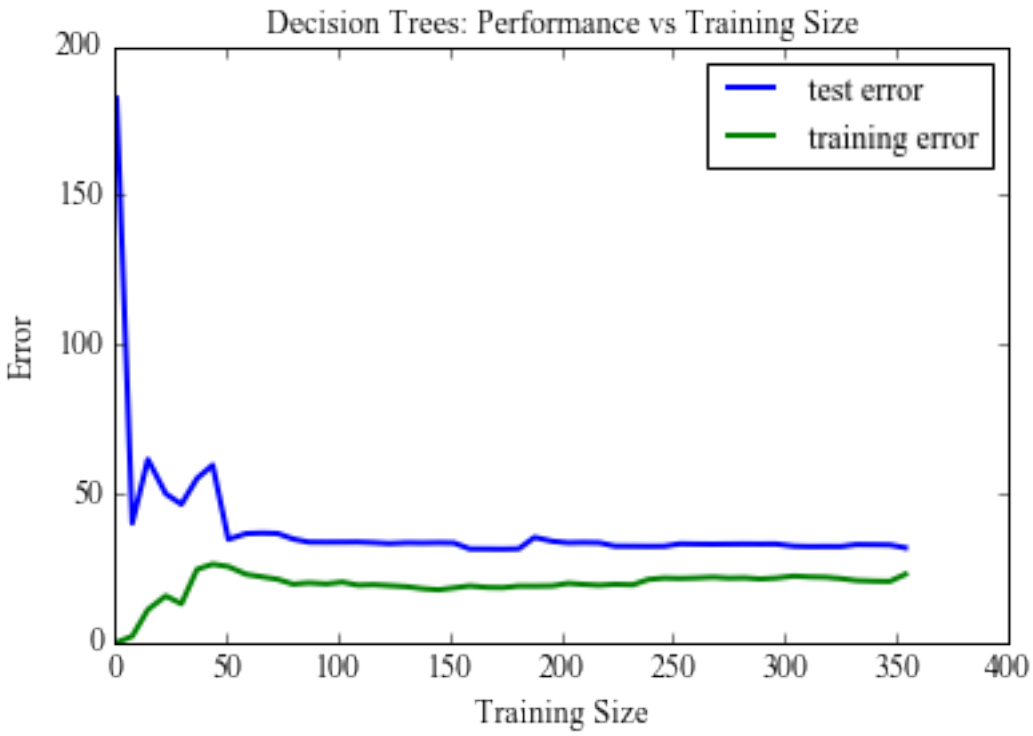
```
        # 2. We will use grid search to fine tune the Decision Tree Regressor and
        # obtain the parameters that generate the best training performance. Set up
        # the grid search object here.
        # http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html#s

        reg = grid_search.GridSearchCV(regressor, parameters,
                                       make_scorer(mean_squared_error, greater_is_better = False))

        # Fit the learner to the training data to obtain the best parameter set
        print "Final Model: "
        print reg.fit(X, y)
        print reg.best_params_

        # Use the model to predict the output of a particular sample
        x = [11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00,
             1.385, 24, 680.0, 20.20, 332.09, 12.13]
        y = reg.predict(x)
        print "House: " + str(x)
        print "Prediction: " + str(y)



    def main():
        """Analyze the Boston housing data. Evaluate and validate the
        performanance of a Decision Tree regressor on the housing data.
        Fine tune the model to make prediction on unseen data."""

        # Load data
        city_data = load_data()

        # Explore the data
        explore_city_data(city_data)

        # Training/Test dataset split
        X_train, y_train, X_test, y_test = split_data(city_data)

        # Learning Curve Graphs
        max_depths = [1,2,3,4,5,6,7,8,9,10]
        for max_depth in max_depths:
            learning_curve(max_depth, X_train, y_train, X_test, y_test)

        # Model Complexity Graph
        model_complexity(X_train, y_train, X_test, y_test)

        # Tune and predict Model
        fit_predict_model(city_data)


    main()
```

```
Size of data (number of houses): 506
Number of features: 13
Minimum price: 5000.0 USD
Maximum price: 50000.0 USD
```

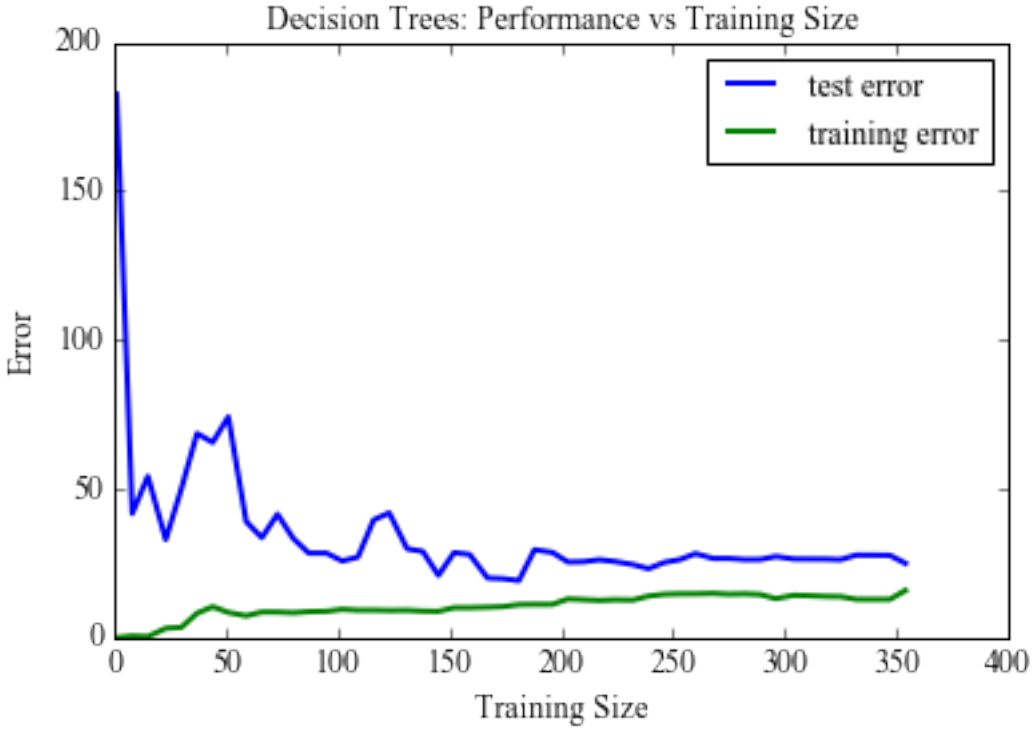
```
Mean price: 22532.8063241 USD
Median price: 21200.0 USD
Standard deviation of house prices: 9188.01154528 USD
Decision Tree with Max Depth:
1
```



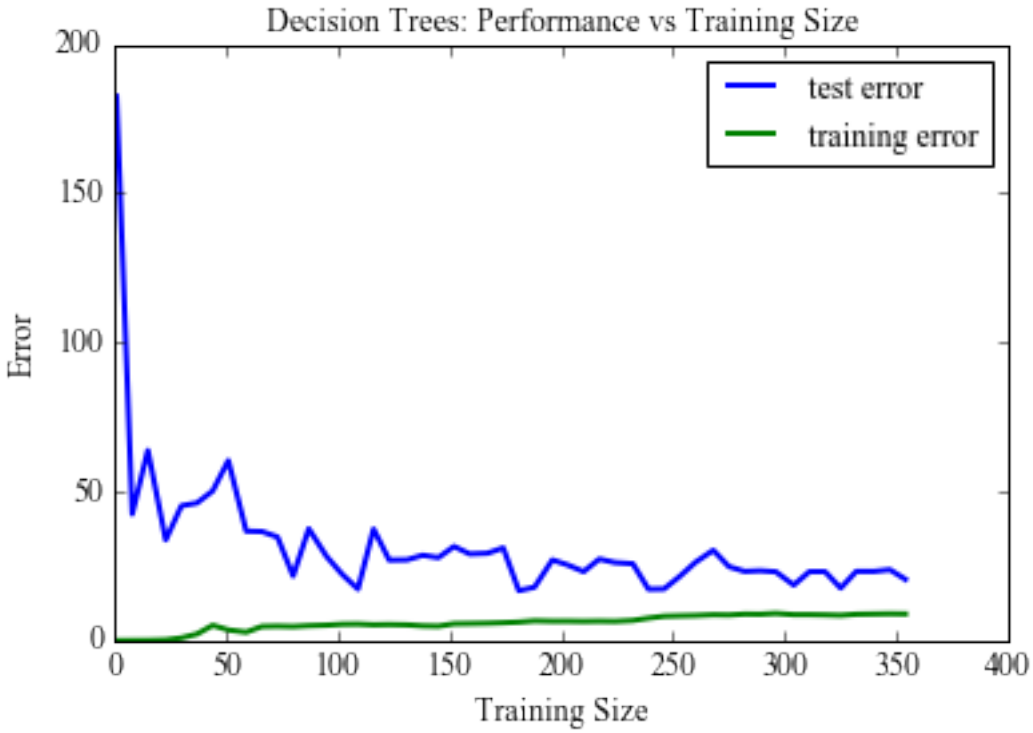

```
Decision Tree with Max Depth:
2
```



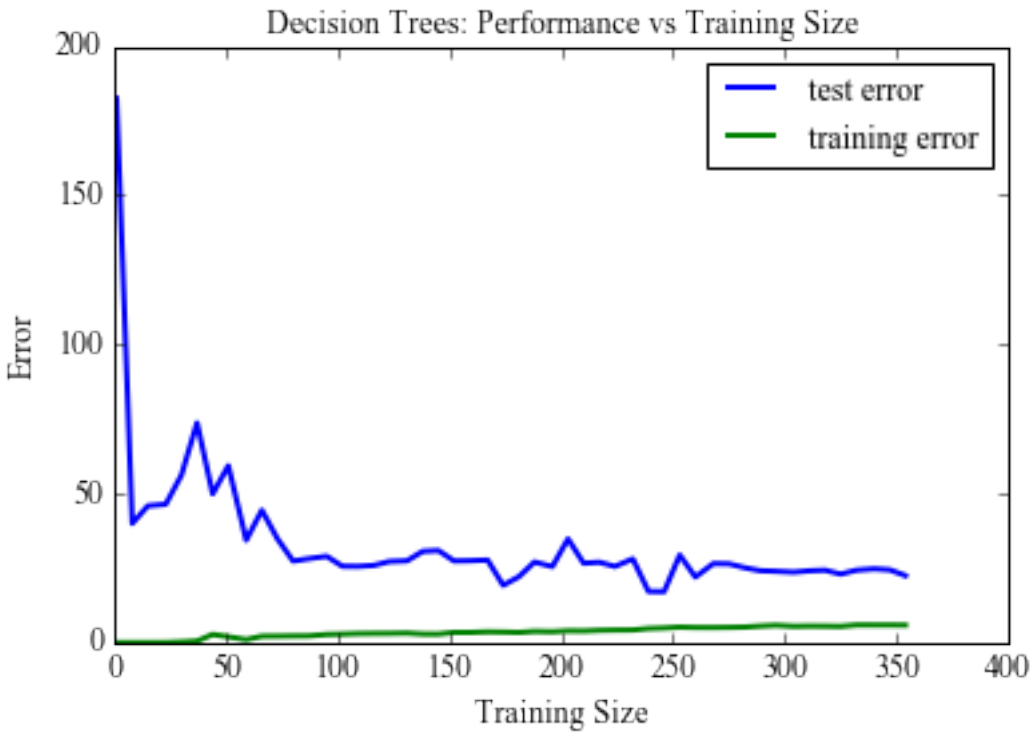

```
Decision Tree with Max Depth:
3
```



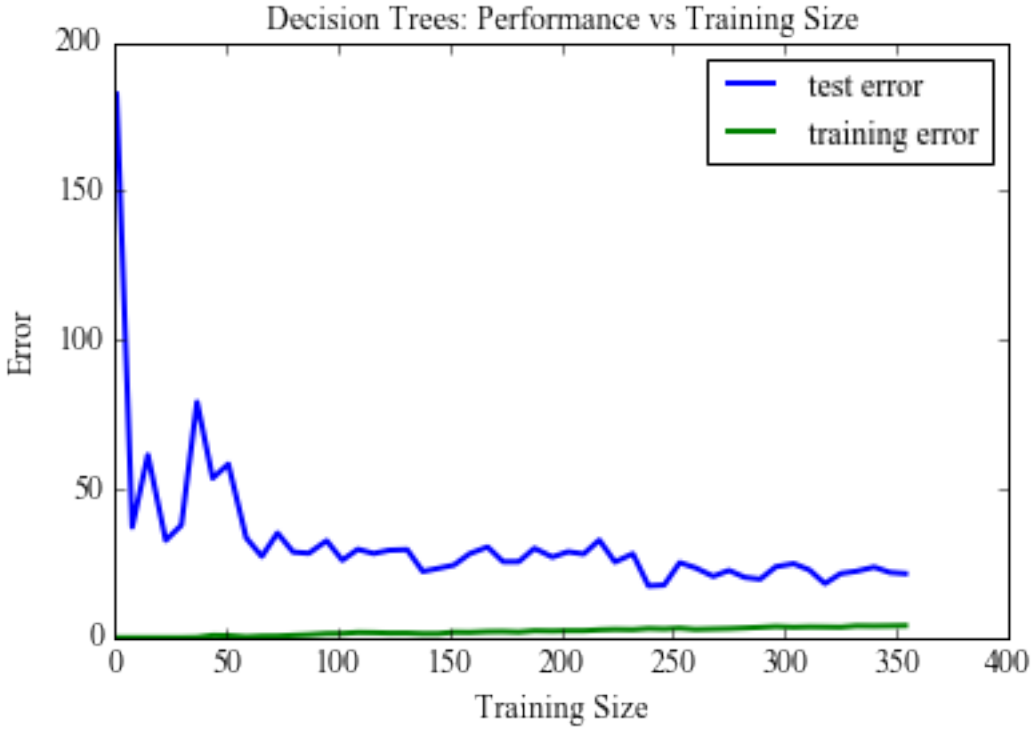
```
Decision Tree with Max Depth:
4
```



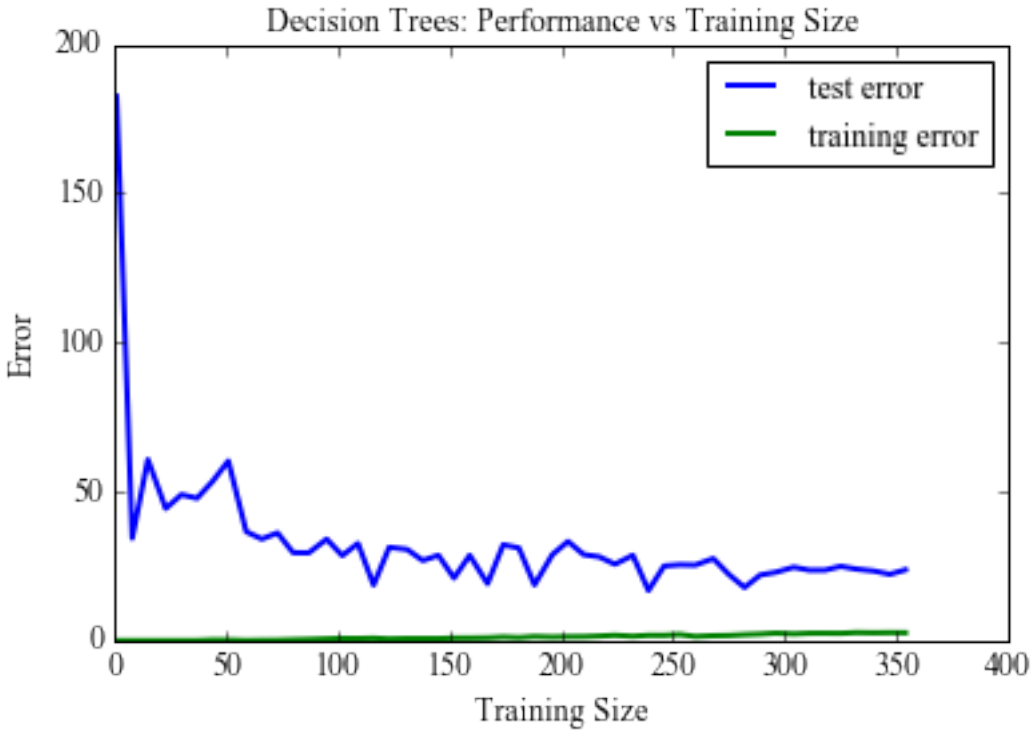

```
Decision Tree with Max Depth:
5
```



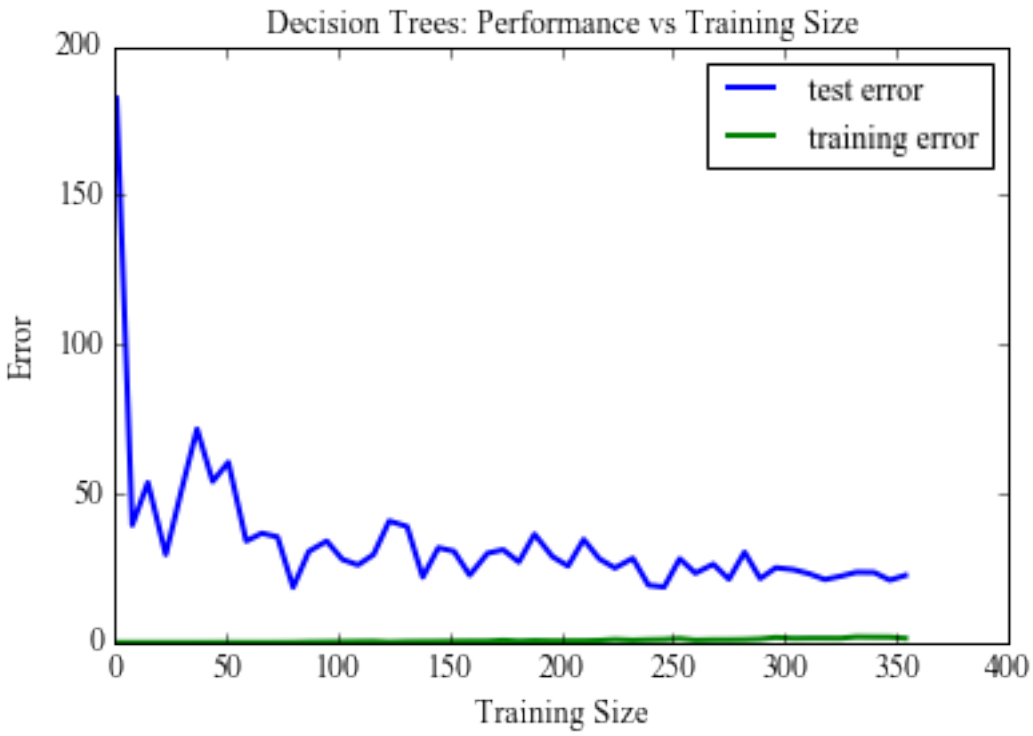

```
Decision Tree with Max Depth:
6
```



```
Decision Tree with Max Depth:
7
```



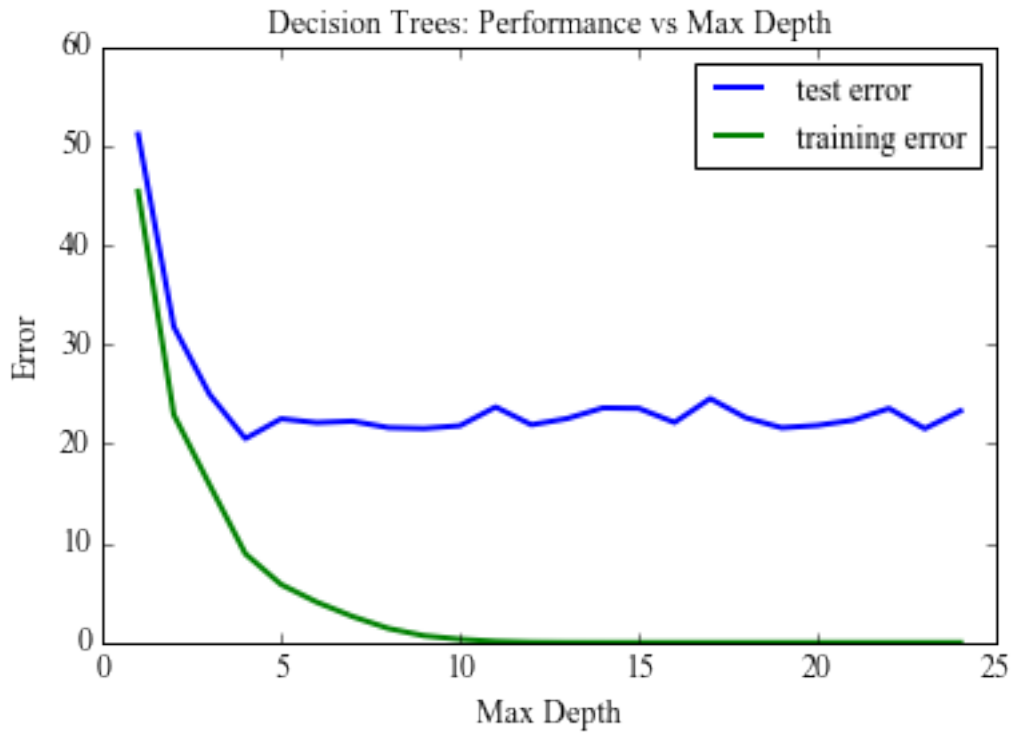

```
Decision Tree with Max Depth:
8
```




```
Decision Tree with Max Depth:
9
```

Decision Tree with Max Depth:
10


Model Complexity:




```
Final Model:
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, presort=False, random_state=None,
           splitter='best'),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'max_depth': (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)},
       pre_dispatch='2*n_jobs', refit=True,
       scoring=make_scorer(mean_squared_error, greater_is_better=False),
       verbose=0)
{'max_depth': 4}
House: [11.95, 0.0, 18.1, 0, 0.659, 5.609, 90.0, 1.385, 24, 680.0, 20.2, 332.09, 12.13]
Prediction: [ 21.62974359]
```